MASTER EN SCIENCES INFORMATIQUES

Traduction (semi-)automatique des répertoires du système radar

Jamme, D.

Award date:
1987

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

TRADUCTION (SEMI-)AUTOMATIQUE DES REPERTOIRES DU SYSTEME RADAR.

Promoteur : J. FICHEFET.

Mémoire présenté par
D. Jamme
en vue de l'obtention du
titre de Licencié et
Maître en Informatique.

Année académique 1986-1987

Remerciements

Je tiens avant tout à remercier vivement Maryline Salmon, qui, grâce à son précieux travail, ainsi qu'à sa constante disponibilité, a permis la réalisation de ce mémoire; tout ce qui concerne les recherches linguistiques (lexiques et grammaires...) est le résultat de ses patientes recherches.

Je remercie également Jacques Paris, dont les remarques et les conseils judicieux m'ont permis de raffiner le contenu de ce travail.

Je tiens enfin à remercier Monsieur Fichefet, promoteur de ce mémoire, pour ses encouragements, ainsi que la grande liberté qu'il m'a laissé quant à la façon de traiter ce sujet.

PARTIE I. INTRODUCTION

PARTIE II. LA TRADUCTION ANGLAIS-ESPERANTO

PARTIE III. LA TRADUCTION ESPERANTO-FRANCAIS

Chapitre 7 : Présentation générale

7.1. Etapes de la traduction
7.2. Exemple
7.3. Travail réalisé

Chapitre 8 : Le dictionnaire Esperanto-français

8.1. Introduction
8.2. Présentation
8.3. Exemple de rubrique
8.4. Conventions
8.5. Règles syntaxiques

Chapitre 9 : Choix du modèle syntaxique

9.1. Petit historique
9.2. Grammaire générative et transformationnelle
  9.2.1. Le modèle d'analyse syntaxique
  9.2.2. La notion de transformation
  9.2.3. Structure de surface et structure profonde
  9.2.4. Critique
9.3. La grammaire de dépendance
  9.3.1. Introduction
  9.3.2. Présentation
  9.3.3. La notion de connexion
  9.3.4. Rapports de dépendance
  9.3.5. Stemma
  9.3.6. Ordre structural et ordre linéaire
  9.3.7. Valence
  9.3.8. Nucléus
  9.3.9. Remarque

Chapitre 10 : Les Augmented Transition Networks ou ATN

10.1. Introduction
10.2. Automates à états finis
10.3. Transition Networks

Chapitre 13 : La génération du texte français

"Translating involves much more than finding corresponding words between two languages. In fact, the words are only minor elements in the total discourse."

Eugen Nida.

Partie I :

Introduction

Chapitre 1 : Contexte du mémoire

Ce mémoire a été réalisé dans le cadre de l'ASBL ARCHIMEDE [Association de ReCHerche en Informatique MEDicalE], formée par des chercheurs composés d'une part de médecins, et d'autre part de professeurs de diverses disciplines scientifiques; leur objectif est d'étudier le plus objectivement possible les mécanismes sous-tendant l'action thérapeutique dans l'application des médecines "douces", ainsi que les résultats obtenus après prescription des remèdes.

L'homéopathie étant la médecine parallèle la plus largement "acceptée" par le grand public, ces chercheurs ont décidé d'y consacrer la plus grosse partie de leurs recherches.

1.1 Homéopathie

Nous rappellerons tout d'abord que l'homéopathie se base sur la Loi de la Similitude, énoncée par Hahnemann vers 1810 [HAHNEM].

Dans le cadre de la doctrine hahnemanienne, la démarche intellectuelle de l'homéopathe comporte 2 phases :

- une individualisation au niveau du malade, c'est à dire la répertorisation de tous les symptômes qu'il présente.

- une individualisation au niveau du remède, qui consiste à trouver dans la nature la drogue (substance végétale, animale ou minérale), capable de provoquer chez un individu sain, les mêmes symptômes que ceux présentés par le malade. D'après la loi de la similitude, cette drogue est censée guérir le malade si elle lui est administrée à des doses infinitésimales [RADAR].

Le médecin est aidé dans sa recherche par des pathogénésies - tableaux de symptômes qui sont apparus chez un individu sain, à qui on a administré des substances ou des drogues-, qui ont été regroupées dans différentes encyclopédies, en plusieurs volumes, appelées Matières Médicales [HERING][ALLEN].

Cependant, la clé d'accés à ces Matières Médicales étant le remède et non pas le symptôme, certains auteurs ont eu l'idée de constituer des répertoires de symptômes, dont les plus célébres sont ceux de KENT [KENT], et le SYNTHESIS qui est un répertoire créé récemment avec le concours d'homéopathes, et comporte des informations de nombreuses origines (Schmidt, Barthel, Clarke,...).

Dans cette optique, Archimède développe et commercialise depuis plusieurs années le logiciel RADAR [Rapid Aid to Drug Aimed Repertorization], qui facilite l'accés aux répertoires de Kent et Synthesis [cfr. 1.2], ainsi qu'aux Matières Médicales, et aide le médecin homéopathe au diagnostic remédial.

Radar peut être qualifié en fait de système-expert à base de connaissances; il comporte un système navigationnel qui permet au médecin de trouver rapidement, et de sélectionner, des symptômes dans des répertoires homéopatiques, et de choisir le remède approprié au patient qui les présente, sur base de méthodes d'aide à la décision multicritère. Il permet aussi au praticien de consulter instantanément les Matières Médicales sur un remède particulier, et de confronter efficacement entre eux différents remèdes.

1.2. Quelques notions sur les répertoires [RADAR]

Ces répertoires sont divisés en chapitres (par exemple, selon les différentes parties du corps humain : Head, Mind,...). [cfr. fig 1.1]

Ils sont structurés selon une hiérarchie d'énoncés de symptômes formant une arborescence dont le répertoire lui-même est la racine; à partir de cette racine, et en collectant les énoncés contenus dans les noeuds successifs d'une même branche, on construit peu à peu l'énoncé complet d'un symptôme.

Une liste de remèdes est rattachée à tout noeud de l'arborescence; on obtient donc la liste des remèdes d'un symptôme en collectant toutes les listes de remèdes associées aux noeuds allant de la racine au noeud-symptôme considéré.

Ces listes de remèdes deviennent de plus en plus petites chaque fois que l'utilisateur descend d'un niveau dans l'arborescence, c'est à dire quand il passe d'un sommet à l'un de ses successeurs.

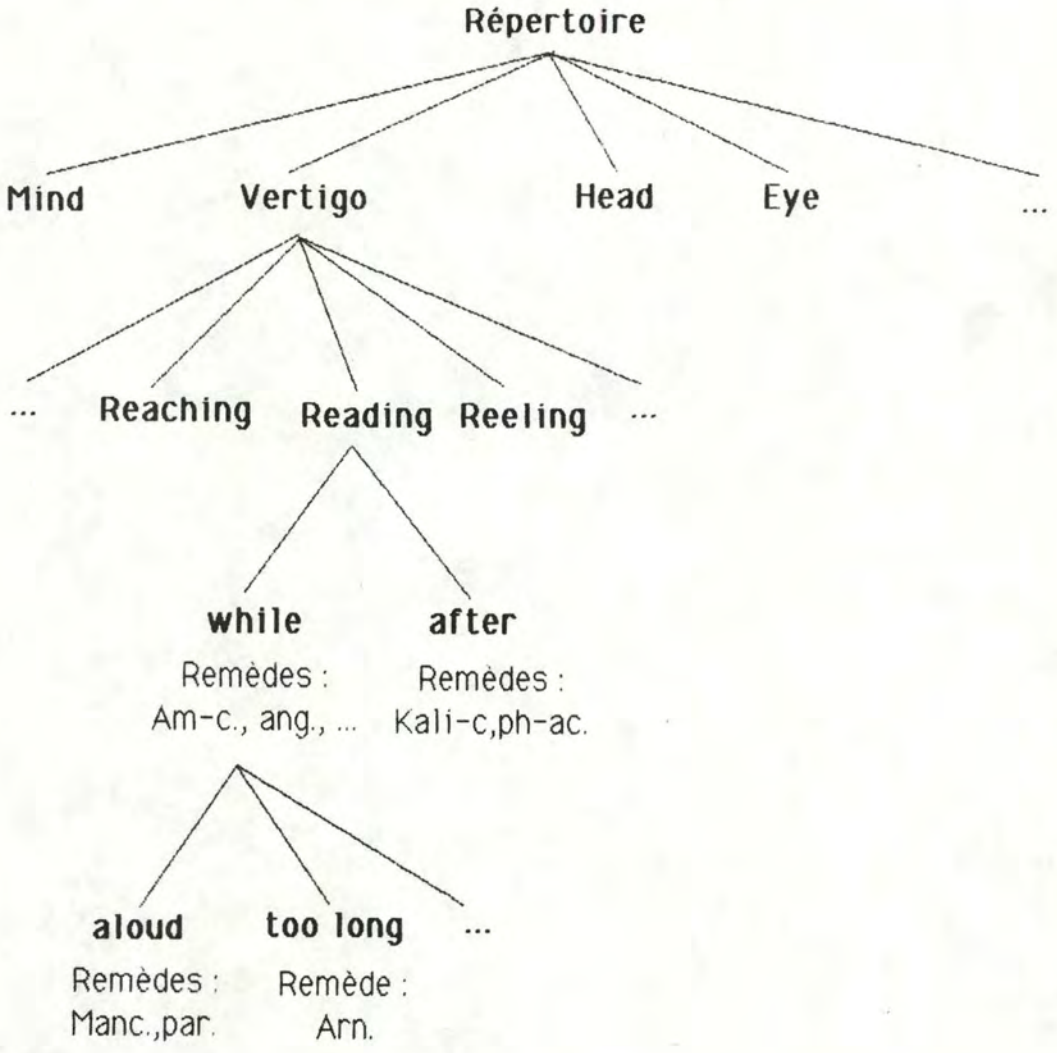


fig 1.1 Exemple de répertoire tiré du Kent

En partant du chapitre "vertigo", on peut trouver la rubrique "reading" qui peut être précisée :

-d'une part, par la rubrique "while" qui peut elle-même être précisée par exemple, par la rubrique "aloud" , ce qui donne vertigo/reading/while/aloud et qui signifie à peu près : "vertiges quand on lit à haute voix".

-d'autre part, par la rubrique "after" qui donne : vertigo/reading/after et qui signifie : "vertiges après avoir lu ".

L'utilisateur peut évidemment s'arrêter quand il le veut, et se contenter par exemple de : vertigo/reading/while.

Les deux répertoires couvrent environ 1574 remèdes et 60000 symptômes.

En conclusion, on peut dire que les répertoires permettent à l'homéopathe de choisir rapidement le remède correspondant aux symptômes observés, tandis que les Matières Médicales l'aident à raffiner et à confirmer le choix du remède. Ce travail du médecin est appelé répertorisation.

1.3 Pourquoi traduire?

Le but principal de la traduction est de permettre aux homéopathes qui ne maîtrisent pas parfaitement la langue anglaise de pouvoir utiliser pleinement le système RADAR, au point de saisir toutes les nuances et les subtilités contenues dans les énoncés de symptômes. En effet, le répertoire de KENT est écrit uniquement en anglais, et le Synthésis est en anglais sous-titré en français et en allemand.

L'objectif du travail réalisé dans le cadre de ce mémoire est donc de sous-titrer le texte de ces deux répertoires dans différentes langues européennes (d'abord français et allemand pour le KENT, puis néerlandais, espagnol, italien, etc... pour les deux).

Il s'agit bien de sous-titrer les répertoires car il n'est pas question de modifier leur organisation selon la nouvelle langue; en effet, les homéopathes font souvent référence à ces répertoires de façon très précise, au numéro de page près, et on ne peut donc se permettre de bouleverser leur présentation qui n'a pas été modifiée depuis leur première édition.

En outre, une version complètement traduite n'aurait plus le support du texte anglais, et donc devrait être parfaitement fiable et posséder le contenu sémantique exact des répertoires anglais; vu l'état actuel de la situation en traduction automatique [cfr. Chap.2], nous avons préféré ne pas trop croire aux miracles...

Chapitre 2 : La situation en traduction automatique.

2.1. Evolution de la traduction automatique.

La possibilité de faire traduire par une machine a été envisagée il y a de nombreuses années (dès 1933); vers 1947, on produisait les premiers programmes de traduction, et en 1954, on présentait au public la première traduction par ordinateur...

Aujourd'hui, bien que les recherches aillent bon train un peu partout dans le monde, et qu'ici et là, on dispose de petits systèmes ou d'outils informatiques capables de réaliser des traductions (dans un domaine bien particulier, et selon des règles très strictes et bien spécifiées), dans la plupart de ces systèmes, et après des années de travail, la qualité de la traduction est encore loin d'être suffisante.

Il faut d'abord rappeler que les connaissances extra-linguistiques (le contexte,...) et la mémoire jouent un rôle tout aussi important que les structures linguistiques dans la traduction. Et le problème de la plupart des systèmes fut précisément qu'ils ne tenaient pas compte de ces connaissances.

Cela fait que le texte obtenu contient généralement toujours des erreurs grossières donnant lieu à des non-sens. Ces erreurs nécessitent une correction par un traducteur humain, dont la fonction demeure donc toujours indispensable; ce travail du traducteur est appelé "post-édition".

On considére généralement qu'une traduction est acceptable si celui qui est chargé de sa révision n'est pas tenté de la refaire lui-même.

Les raisons pour lesquelles il reste toujours des erreurs sont dues à la complexité du langage humain et des processus de transposition.

Si la traduction littérale est essentiellement une affaire de lexique, la traduction non-littérale est une affaire de stylistique...

Or, les régles de réécriture d'une langue dans une autre sont difficiles à introduire dans la mémoire d'un ordinateur, ce qui fait que la traduction automatique ne pourra pendant longtemps encore être utilisée avec des résultats satisfaisants, que pour des textes répondant à des critéres bien déterminés; ce sont les textes dont la structure syntaxique est simple, et la terminologie non-équivoque.[LAVOREL]

Aussi, aujourd'hui, et en dépit des progrés réalisés, il semble pratiquement évident que les systèmes de traduction automatique ne resteront encore longtemps que des compléments utiles aux traducteurs humains, et qu'ils ne remplaceront jamais ces traducteurs. Ceux-ci demeureront nécessaires, pour améliorer ce qu'auront produit les sytèmes de traduction automatique, c'est pourquoi on devrait plutôt parler de "traduction humaine assistée par ordinateur" au lieu de traduction automatique, et de "système d'aide à la traduction" au lieu de système de T.A...

2.2. Outils informatiques d'aide à la traduction

2.2.1. Traitements de textes intelligents

Un traitement de texte peut être utile aux traducteurs à condition d'être adapté à leurs exigences particulières : écran suffisamment large, facilité de manipulation simultanée de plusieurs textes, fonctions d'usage facile, jeu de caractères étendu immédiatement accessible...

Il s'agit de toute façon d'un préalable obligatoire pour celui qui veut utiliser d'autres outils informatiques.

2.2.2. Bases de données terminologiques

Le traducteur humain a toujours nécessairement besoin d'un bon dictionnaire sous la main, car une grosse partie de son travail (jusque 70 %) consiste à consulter des dictionnaires; or, cette recherche prend du temps, et on a eu l'idée de créer des bases de données terminologiques, qui sont en fait des dictionnaires automatisés, et qui contiennent les traductions de tout mot, ainsi qu'un ensemble d'informations sur ce mot. [cfr. Chap.4]

On a ensuite créé un ensemble de programmes permettant d'accéder facilement à ces bases de données.

Exemple : Eurodicautom à la CEE [EURO].

2.2.3. Autres programmes utiles

Un certain nombre d'autres programmes peuvent aider le traducteur, depuis le simple vérificateur orthographique de textes, jusqu'au programme capable de

signaler qu'une phrase donnée est d'une complexité grammaticale trop importante par rapport au niveau de complexité moyen des phrases habituellement rencontrées dans un certain type de textes.

2.3. Notions de traduction automatique

2.3.1 L'analyse d'un texte

a) Contenu d'un texte

On peut considérer qu'un texte a un double contenu :

- un contenu syntaxique : caractéres organisés suivant des règles grammaticales.
- un contenu sémantique : c'est la signification du message associé au texte.

Le contenu syntaxique dépend de la langue dans laquelle le texte est exprimé; au sein d'une même langue, il dépend de règles grammaticales pouvant varier selon les régions, les communautés, les contextes de discours, etc...

Le contenu sémantique est relativement indépendant de la langue utilisée; dans une même langue, il est relativement indépendant du support (écrit, parlé,...).

b) Limites des machines

L'homme et la machine n'ont pas la même conception des textes : un ordinateur n'a pas accès au contenu sémantique, il ne peut pas comprendre une idée. Il accède uniquement au contenu syntaxique, à l'aide de programmes capables de manipuler des règles grammaticales.

De plus, pour être accessible par un ordinateur, une grammaire doit être **formalisée** :

- elle doit posséder un nombre fini de règles (pour tenir en machine).
- ses règles doivent être complètes (pour décrire toutes les phrases de la langue).
- ses règles doivent être cohérentes (pour ne décrire que les phrases de la langue).

Or, la construction de grammaires formalisées de la langue naturelle pose un certain nombre de problèmes.

c) Exemples de problèmes [RUWET]

- Exemple 1 : structure syntaxique différente malgré l'aspect extérieur.

"La circulation sur la grand-route a été déviée par la Gendarmerie"
"La circulation sur la grand-route a été déviée par un chemin de campagne"

-Exemple 2 : structure syntaxique semblable malgré l'aspect extérieur.

"Jacques aime Marie"
"Le petit vieillard aux cheveux blancs qui habite en face de chez nous a perdu ses lunettes"

- Exemple 3 : ambiguïté syntaxique (phrases qui peuvent être comprises de plusieurs manières différentes).

"Jacques aime mieux Marie que Simone"
"Je lui ai proposé d'écrire un texte"
"J'ai écouté le disque de Sardou"
"J'ignore quels ennemis redoutaient les soldats"

- Exemple 4 : relations entre certains éléments de phrases.

La relation entre Jacques et Marie est identique dans :
"Jacques aime Marie" et "Marie est aimée de Jacques",
mais pas dans "Marie aime Jacques"...

2.3.2. Le processus de traduction

En considérant un texte comme le support d'un double contenu syntaxique et sémantique, et en admettant que seule la syntaxe dépend de la langue utilisée, le processus de traduction peut se schématiser comme suit :

1. Dégager la structure syntaxique du texte à traduire, grâce aux règles d'analyse construites sur base de la grammaire.

2. En tirer le contenu sémantique grâce à des règles d'interprétation.

3. Construire une structure syntaxique qui corresponde à la langue-cible et qui respecte le contenu sémantique du texte.

4. Générer le texte en langue-cible à partir de la structure syntaxique.

Cette démarche se retrouve dans la plupart des techniques de traduction qui ont été réalisées à ce jour.

2.3.3. Les techniques de traduction automatique

a) Les premiers systèmes "language pair"

Ces "monstres" virent le jour à une époque où l'on croyait qu'il suffisait d'une grammaire suffisamment détaillée et d'un dictionnaire suffisamment riche pour traduire un texte, sans trop se préoccuper de l'aspect sémantique.

Leur principe de fonctionnement est le suivant : analyse lexicale phrase par phrase du texte source, transformation de la structure découverte, et remplacement des différents mots par leur traduction; il s'agit donc d'une traduction "mot à mot".

Ils se caractérisent par leur structure massive, l'absence de modularité, et un nombre impressionnant de routines spécialisées ajoutées à chaque nouvelle erreur de traduction découverte.

La qualité des traductions est généralement mauvaise, et ces systèmes ne sont pratiquement plus utilisés ni développés aujourd'hui.

b) La traduction par transfert de représentation

La traduction par transfert de représentation consiste à analyser le texte en langue-source pour en découvrir la structure profonde, à élaborer une représentation interne sophistiquée de cette structure en langue-source, puis à procéder à un transfert de représentation pour obtenir une représentation interne du texte en langue-cible; enfin, cette représentation est générée en langue-cible.

La traduction est ici exclusivement bilingue, et si on veut traduire des textes dans m langues-cibles à partir de n langues-sources, il faut n*m composants de transfert de représentations internes, ainsi que n analyseurs et m générateurs [cfr. fig. 2.1].

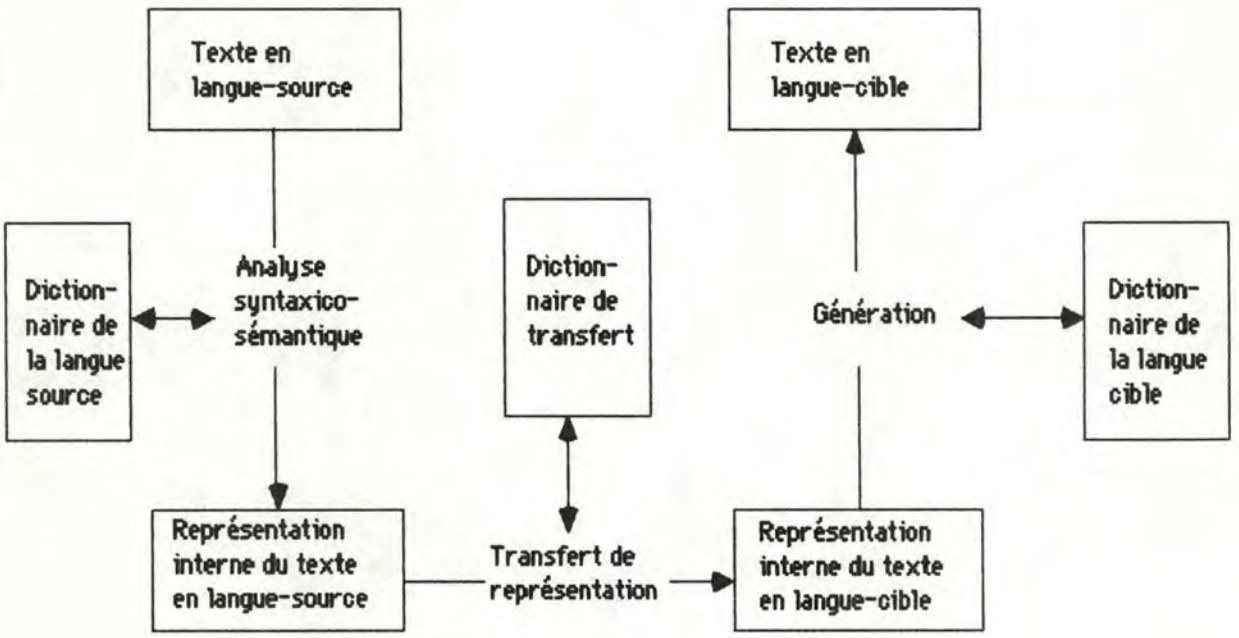


fig. 2.1 : Traduction par transfert de représentation

c) La traduction via un interlangage

Le principe de la traduction avec interlangage est de generer lors de l'analyse du texte en langue-source, une représentation interne dans une langue intermédiaire universel, qui exprime le sens du texte et les dépendances entre les divers composants. Ensuite, le texte en langue-cible est généré directement à partir de cette représentation.

Ces systèmes sont plus adaptés à la traduction multilingue : une fois la représentation élaborée, il ne reste plus qu'à en tirer un équivalent dans la langue-cible; il suffit donc, pour n langues-sources, et m langues-cibles, d'implémenter n analyseurs et m générateurs [cfr. fig.2.2].

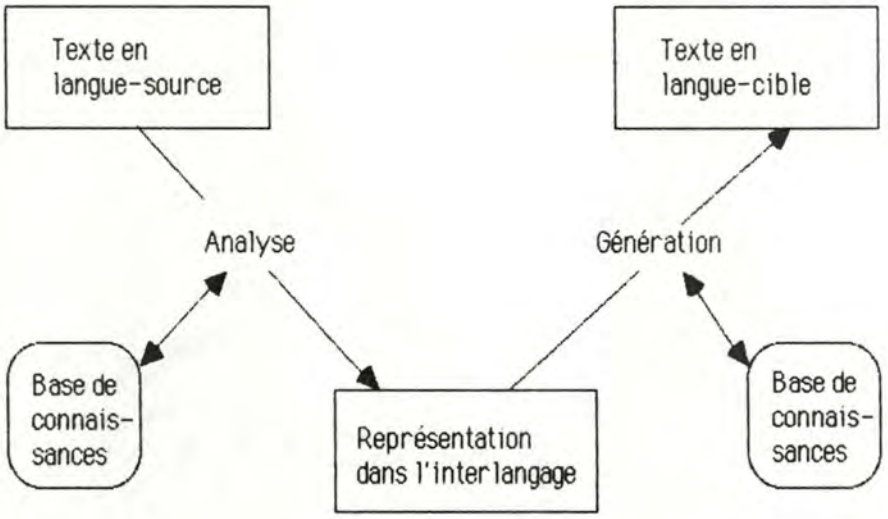


fig.2.2 : La traduction via un interlangage

Chapitre 3 : Choix d'un système de traduction

3.1. La technique choisie

Le but du travail n'est pas de construire un nouveau répertoire qui serait une traduction séparée de l'original; comme nous l'avons expliqué au chapitre 1 [cfr. 1.3], il s'agit de sous-titrer les répertoires sous une forme similaire à celle du SYNTHESIS : un texte anglais, sous-titré de traductions dans les langues-cibles.

En effet, à l'origine [HOGNE], ce projet semblait réalisable tel quel, en raison de la structure des répertoires : univers de discours restreint, contexte des phrases toujours connu, vocabulaire relativement spécifique, peu de mots à double-sens, et syntaxe particulière (phrases courtes, de style télégraphique, souvent abrégées,...).

Après une comparaison des différentes techniques de traduction automatique existantes, il fut décidé de retenir la technique de traduction via un interlangage, et de choisir l'Esperanto comme interlangage [HOGNE].

3.2. Présentation de l'Esperanto

3.2.1. Historique

L'Esperanto est une langue que je qualifierais de "semi-artificielle".

Elle est artificielle, parce qu'entièrement voulue, créée par un seul homme, le Docteur Lazare Louis Zamenhof.

Zamenhof est né à Byalistok en 1859. Sa ville fait alors partie de l'empire russe -elle est maintenant polonaise- et connaît une convergence de différentes populations, c'est à dire de différentes cultures et langues. On y parle russe, allemand, polonais, français... Zamenhof souffre d'autant plus de cette disparité qu'il vit dans le ghetto juif, et est donc doublement victime des inévitables tensions qui naissent entre ces différents groupes sociaux et culturels.

C'est cette situation qui est à l'origine de la création de la langue Esperanto (Esperanto est le pseudonyme sous lequel Zamenhof a publié sa première brochure); Zamenhof voulait une langue de "réconciliation entre les hommes", un outil de communication pure, qui ne serait pas entaché de connotations culturelles ou politiques. Un langage qu'il fallait donc inventer de toutes pièces.

L'organisation, la structure de cette langue devait être adaptée à ce but de communication universelle. Il a donc privilégié une structure logique, régulière, une prononciation fixe de chaque lettre, un lexique le plus "international" possible. Mais tout ceci sans perte de subtilité dans l'expression.

Autre point important : Zamenhof a conçu des règles telles que l'on puisse continuer à développer sa langue après lui. Ce qui fait qu'on dispose aujourd'hui d'une terminologie importante dans divers domaines scientifiques.

Nous qualifions la langue de "semi-artificielle", parce que malgré cet aspect construit, la langue s'étudie comme toute autre langue naturelle et a d'ailleurs donné lieu à une littérature originale.

3.2.2. Syntaxe

L'Esperanto est basé sur seize règles de grammaire qui suffisent à une expression correcte et complète.

Exemple:

- Un seul article invariable : la
- Tous les substantifs se terminent par o
- Tous les adjectifs se terminent par a
- Tous les adverbes se terminent par e
- Il n'existe qu'une seule marque du pluriel : j
- Tous les verbes à l'infinitif se terminent par i
- Tous les verbes au présent se terminent par as
- Tous les verbes au passé se terminent par is
- Tous les verbes au futur se terminent par os

Ces marques grammaticales viennent s'ajouter aux radicaux des mots, sans altération aucune, c'est à dire aucun changement de voyelles, redoublement de consonnes..., comme cela arrive très fréquemment dans les langues naturelles, où l'on change parfois même le radical entièrement.

Ex : Le verbe aller en français : je vais
tu vas
nous allons
nous irons...

Les formes correspondantes en Esperanto sont tout à fait régulières :
mi iras
vi iras
ni iras
ni iros...

Pour ce qui est de la structure de la phrase, elle ne diffère pas tellement de celle du français. Avec un avantage : chaque catégorie grammaticale étant clairement marquée, on bénéficie de plus de liberté quant à la position des mots dans la proposition.

exemple 1 : La knabino estas tre beleta (la fille est très jolie).

Les deux groupes soulignés peuvent changer de place sans problème.

exemple2 : El Namuro mi venas. (je viens à Namur).

Ces trois groupes peuvent permuter.

3.2.3. Lexique

Le lexique est composé de racines invariables qui en sont les unités signifiantes, et de toute une série d'affixes qui modulent le sens premier des racines.

Ces racines, ou lexèmes, sont tirés à 75% des langues latines (latin et français) et 20 % des langues anglo-germaniques, le reste est constitué d'emprunts au grec, aux langues slaves...
Les lexèmes latins sont les plus internationaux : 40 % d'entre eux seraient immédiatement compréhensibles par un Russe.

Cette façon de procéder rend beaucoup plus facile l'acquisition du vocabulaire. Tout mot rencontré est décomposable en un certain nombre d'unités signifiantes. Si on connaît le sens du radical, on peut analyser le mot et en déduire sa signification.

ex : Vous connaissez le radical ter-, vous pouvez comprendre :
tero : (la) terre
subtera : sous-terrain
enterigi : enterrer
terpomo : pomme de terre

Comme le montre le dernier exemple, on peut également former des mots composés de plusieurs radicaux en les agglutinant très simplement.
ex : terkulturo : agriculture
terkulturisto : agriculteur

Ces mêmes mots composés peuvent également être modifiés par des affixes.

3.3. Avantages de l'Esperanto

3.3.1. Réduction des ambiguités et des difficultés d'analyse grammaticale.

Grâce à la structure de l'Esperanto, on va pouvoir lever un très grand nombre d'ambiguités ou d'irrégularités des langues naturelles.

Du point de vue des irrégularités, nous avons cité l'exemple du verbe aller en français, qui change plusieurs fois de radical. Idem pour l'anglais : il n'est pas évident que l'ordinateur puisse faire immédiatement le lien entre go et went, have to do et must...

Les ambiguités du langage sont beaucoup plus nombreuses qu'on pourrait le croire à première vue. En effet, nos langues modernes présentent un paradoxe. Nous sommes persuadés qu'une langue comme l'anglais, par exemple, se simplifie au fil de son évolution: de moins en moins de verbes "irréguliers", même terminaison pour différentes fonctions grammaticales, souplesse d'utilisation du vocabulaire... Tout ceci nous rend apparemment l'apprentissage du langage beaucoup plus aisé, notre mémoire semble moins sollicitée.

Mais à y regarder de plus près, et notamment quand on essaie de faire traîter le langage par un ordinateur, on s'aperçoit vite que cette apparente simplicité cache un nombre incalculable d'ambiguités. Ambiguités qui sont toutes des obstacles à un traîtement (une traduction, par exemple) correct des textes soumis à la machine. [cfr. 2.3.1 c)]

Exemples :

Sheep = un mouton ou des moutons. En Esperanto : shafo au singulier et shafoj au pluriel.

Un mot tout simple comme 'after' peut avoir plusieurs significations en anglais.

En Esperanto, on a des mots différents :

conjonction :	post kiam
adverbe :	poste
adjectif :	posta
préposition :	post
	laù
	pri

En effet, la préposition after a trois significations différentes :

after dinner : post mangho
after him, that film was not so good : laù li (selon lui)
what is he after ? (qu'est-ce qu'il cherche?) : pri kio li serchas ?

Une fois traduits en Esperanto ces différents messages seront complètement univoques. L'ordinateur ne se perdra plus en diverses conjectures quant à la fonction de tel ou tel mot. Désambiguation d'autant plus précieuse quand on a affaire à des mots tels que : get, let, have... qui prennent plusieurs pages de dictionnaire.

3.3.2. Traductions multilingues plus aisées

Le fait de pouvoir bénéficier d'une version Esperanto d'un texte sera donc très utile quand nous voudrons retraduire ce texte dans d'autres langues naturelles.

Cette technique de traduction est avantageuse pour traduire des textes de grande diffusion (c'est à dire des textes traduits dans de nombreuses langues différentes).

Le texte Esperanto étant (presque) totalement dépourvu d'ambiguités, la traduction en d'autres langues naturelles devrait pouvoir se faire quasi-automatiquement (sans richesse de style, évidemment).

3.3.3. Manipulation facile du texte Esperanto

Les systèmes informatiques actuels qui manipulent la langue naturelle utilisent des représentations sophistiquées de ces textes qui permettent d'en exprimer le contenu sémantique.

Un des problèmes liés à ces représentations est qu'elles sont peu accessibles à l'homme de façon directe. Pour vérifier la concordance entre un texte et sa représentation sous forme de réseau sémantique, par exemple, il faut disposer de primitives d'accès spécialisées.

Avec l'Esperanto, la représentation interne du texte est simplement un fichier de texte que l'on peut lire normalement... et comprendre si on connait la langue.

3.3.4. Exploitation "secondaire" : accès à la sémantique du fichier.

On a vu que les mots Esperanto étaient constitués de racines agglutinées les unes aux autres. Cette particularité du lexique pourra être exploitée dans les développements futurs du projet RADAR, pour construire des index sur les répertoires dont les clefs ne seraient plus seulement syntaxiques mais sémantiques.

En effet, actuellement, pour rechercher un mot, un symptôme ou une rubrique dans un répertoire, il faut connaître son orthographe exacte, ou du moins celle de ses premières lettres.

Avec un index sur les racines esperanto contenues dans le fichier, et une procédure de conversion en Esperanto d'un mot anglais (par exemple), on pourra rechercher tous les mots du répertoire qui contiennent une racine correspondant au mot entré.

Par exemple, avec la racine ter, on pourra retrouver enterigi (enterrer) et tero (terre) qui se traduisent en anglais par to bury et earth ou ground. Ces mots sont liés sémantiquement, mais le système actuel est incapable de s'en rendre compte.

3.4. Le processus de traduction

3.4.1. Rappels [HOGNE]

Le système doit pouvoir traduire indifféremment un répertoire KENT ou un répertoire SYNTHESIS, bien qu'ils aient des structures différentes.

Cependant, on a décidé que nos traductions seraient de la forme de celles du SYNTHESIS (une ligne anglaise sous titrée par des traductions dans deux langues différentes), et notre système ne travaillera donc que sur des fichiers de type SYNTHESIS; on a donc construit un programme de conversion des fichiers KENT en fichiers SYNTHESIS.

En outre, un autre programme permet de simplifier les fichiers SYNTHESIS afin d'éliminer certaines particularités syntaxiques (rubriques, doubles,...) qui handicapent la mise en oeuvre d'analyseurs syntaxiques "propres".

3.4.2. Etapes de la traduction

La traduction en elle-même se compose en trois étapes :

-Vider le répertoire de ses traductions éventuelles.

-Traduire le répertoire en Esperanto.

-Traduire le répertoire en langue-cible.

[cfr. fig 3.1]

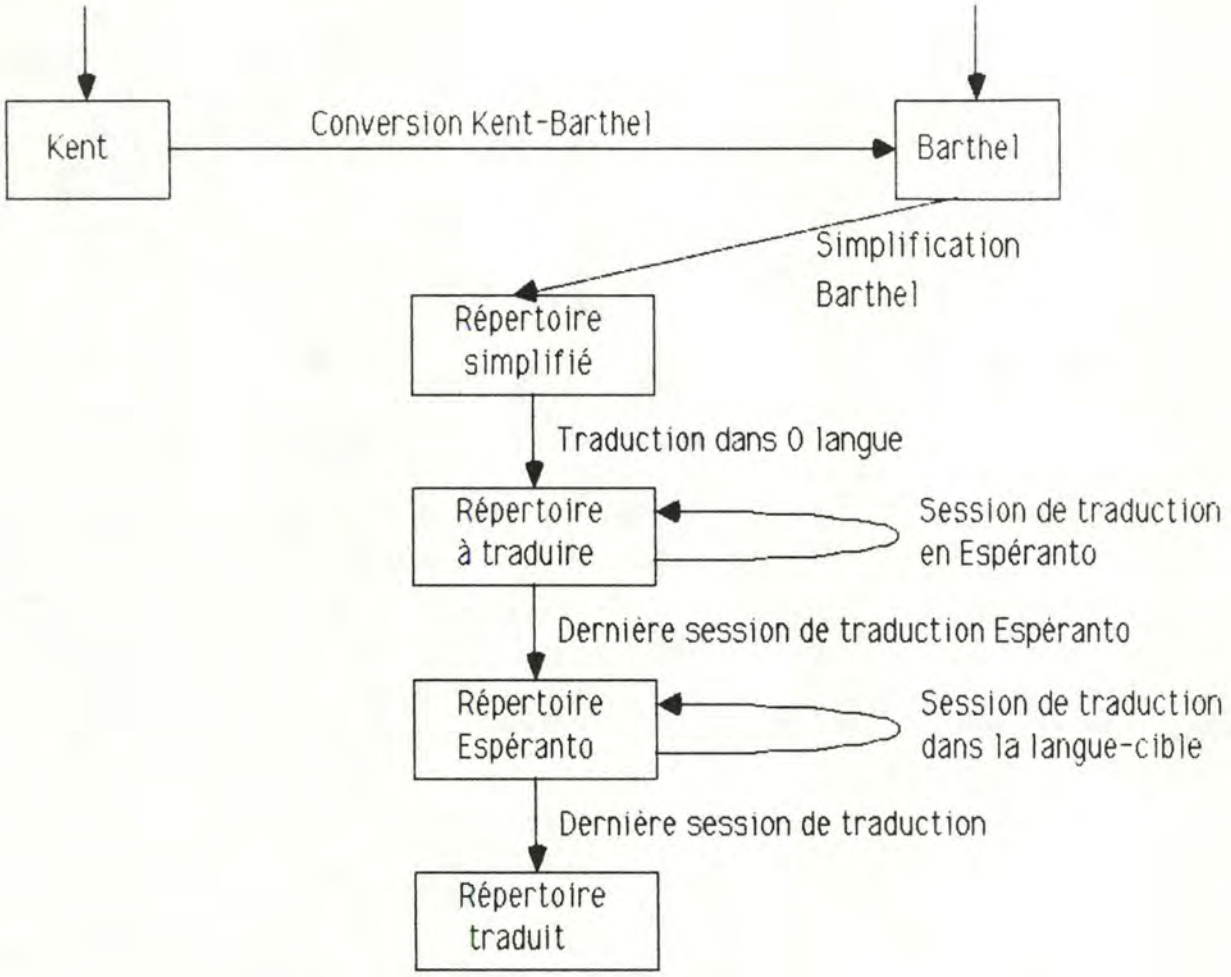


fig 3.1 Etapes de la traduction d'un répertoire

3.4.3. Scénario d'une session de traduction

Une session de traduction d'un répertoire consiste à le parcourir séquentiellement à la recherche des phrases qui ne sont pas encore traduites, à traduire ces phrases, et à itérer le processus jusqu'à ce que le traducteur humain décide de stopper le processus, ou que le répertoire soit complètement traduit.

Ainsi, si une phrase pose problème, on peut postposer sa traduction, et elle sera de nouveau soumise à l'analyse lors de la session suivante.

La mise en oeuvre de ce programme est simple: il parcourt séquentiellemnt le répertoire à traduire, et le recopie au fur et à mesure dans un autre fichier.

Lorsqu'il rencontre une phrase sans traduction, ou avec une seule traduction (celle en Esperanto), il la transmet au module traducteur qui la renvoie avec une traduction; il recopie alors la phrase et sa traduction dans le fichier-résultat, et répète le processus jusqu'à la fin du fichier-source, ou jusqu'à une interruption par l'utilisateur [cfr. fig 3.2].

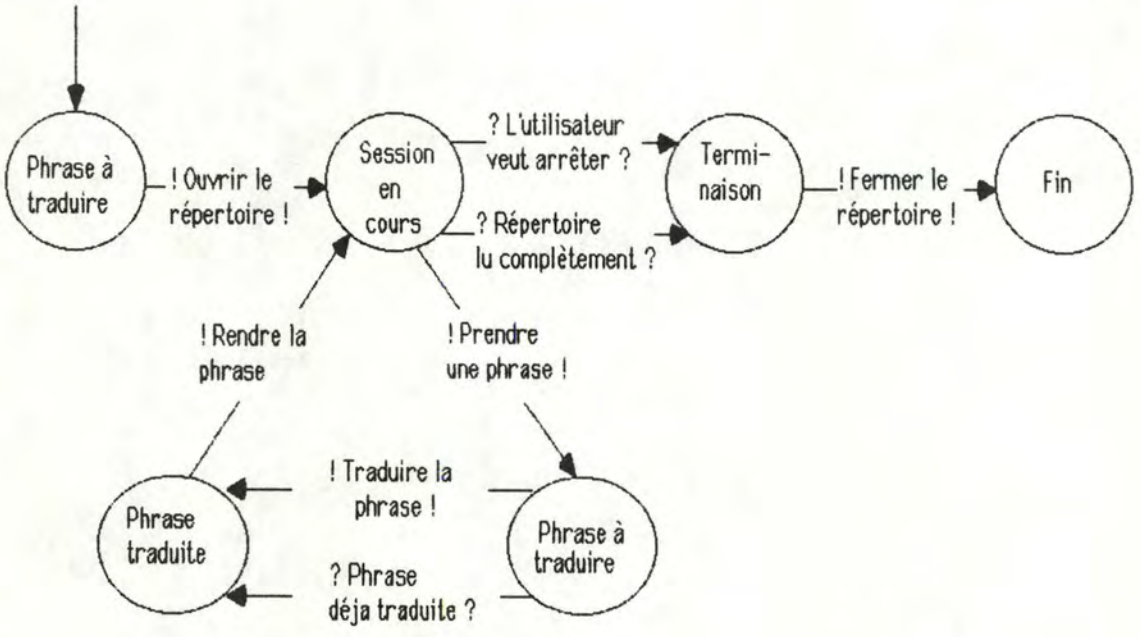


Figure 3.2 : Graphe des états d'une session de traduction

Partie II :

La traduction anglais-Esperanto

Chapitre 4 : Le dictionnaire anglais-Esperanto

4.1. Introduction

Dans toute technique de traduction automatique, il est nécessaire de disposer d'un dictionnaire langue-source - langue-cible.

Dans notre cas, il fallait donc un dictionnaire qui contenait la ou les traductions de tout mot apparaissant dans les répertoires anglais.

Nous sommes donc partis de la liste complète des mots des répertoires, et pour chaque terme nous avons établi ses polysémies, c'est à dire la ou les catégories grammaticales auxquelles il appartient, et pour chaque catégorie, quelle(s) réalité(s) il recouvre. Cela a exigé une recherche minutieuse dans de bons dictionnaires anglais.

Ensuite il a fallu rechercher un correspondant Esperanto pour toute acception, dans des dictionnaires et différentes sources provenant de la Communauté Esperantiste. Il a cependant été nécessaire d'inventer certains termes Esperanto, notamment pour traduire les termes médicaux, car nous ne trouvions de traduction nulle part...

4.2 Présentation

Le dictionnaire est constitué d'une suite de phrases concernant chacune un mot anglais, et tout mot anglais pouvant avoir plusieurs catégories grammaticales, on a repésenté chacune des catégories d'un même mot sur une ligne différente.

A chacune de ces catégories peuvent correspondre une ou plusieurs traductions Esperanto.

Un mot peut aussi intervenir dans des expressions idiomatiques; dans ce cas la traduction de chaque expression est reprise en plus de l'énoncé de l'expression elle-même.

D'autre part, lorsque des mots sont dérivés d'autres mots (par exemple les formes conjuguées des verbes irréguliers), on marque simplement la référence au mot dont ils sont dérivés.

Enfin, pour certaines traductions Esperanto, nous avons ajouté une information explicative (par exemple, pour signaler que ce qui suit doit être à l'accusatif, etc).

4.3. Règles d'encodage

Comme nous l'avons dit ci-dessus, un mot anglais peut être représenté sur plusieurs lignes; dans ce cas, le mot est retranscrit en tête de chaque ligne.

Notons encore qu'afin de distinguer aisément les différentes racines et affixes des mots Esperanto, la première lettre de chaque racine est écrite en majuscule. Ceci sera utile pour l'analyseur lexical [cfr 13.3], et pour l'accès sémantique aux répertoires.
Exemple : Au lieu de malabundeco, on écrit MalAbundEcO.

Le tableau 4.1 reprend tous les caractères utilisés pour l'encodage du dictionnaire.

Pour indiquer	Caractère
un début de description grammaticale	£
un début de traduction Espéranto	=
la séparation entre deux traductions	/
un début d'information sur une traduction	[
un début de renvoi à une autre rubrique	>
le début d'une expression idiomatique	&

tableau 4.1

4.4. Définition syntaxique du dictionnaire

(DICTIONNAIRE) <-- (RUBRIQUE) | (RUBRIQUE)(DICTIONNAIRE)

(RUBRIQUE) <-- (ENTREE)(PARTIE LITTERALE) | (ENTREE)(PARTIE LITTERALE)(PARTIE IDIOMATIQUE)

(ENTREE) <-- (CHAINE)

(PARTIE LITTERALE) <-- (DESCRIPTION LITTERALE) | (DESCRIPTION LITTERALE)(PARTIE LITTERALE)

(DESCRIPTION LITTERALE) <-- **£** (CATEGORIE GRAMMATICALE)(DESCRIPTION)

(CATEGORIE GRAMMATICALE) <-- ADV | V0| V1 | V2 | PREP | ...

(DESCRIPTION) <-- (RENVOI)(DESCRIPTEUR)

(RENVOI) <-- **>** (CHAINE)
(DESCRIPTEUR) <-- = (TRADUCTION) | = (TRADUCTION) **[** (INFORMATION)

(TRADUCTION) <-- (CHAINE) | (CHAINE) **/** (TRADUCTION)

(INFORMATION) <-- **+** ACC | **+** NOM | INTRANS | ...

(PARTIE IDIOMATIQUE) <-- (DESCRIPTION IDIOMATIQUE)(PARTIE IDIOMATIQUE)

(DESCRIPTION IDIOMATIQUE) <-- **&** (CHAINE)(DESCRIPTEUR)

(DICTIONNAIRE) <-- (RUBRIQUE) | (RUBRIQUE)(DICTIONNAIRE)

(RUBRIQUE) <-- (ENTREE)(PARTIE LITTERALE) | (ENTREE)(PARTIE LITTERALE)(PARTIE IDIOMATIQUE)

(ENTREE) <-- (CHAINE)

(PARTIE LITTERALE) <-- (DESCRIPTION LITTERALE) | (DESCRIPTION LITTERALE)(PARTIE LITTERALE)

(DESCRIPTION LITTERALE) <-- £ (CATEGORIE GRAMMATICALE)(DESCRIPTION)

(CATEGORIE GRAMMATICALE) <-- ADV | V0| V1 | V2 | PREP | ...

(DESCRIPTION) <-- (RENVOI)(DESCRIPTEUR)

(RENVOI) <-- > (CHAINE)

(DESCRIPTEUR) <-- = (TRADUCTION) | = (TRADUCTION) [(INFORMATION)

(TRADUCTION) <-- (CHAINE) | (CHAINE) / (TRADUCTION)

(INFORMATION) <-- + ACC | + NOM | INTRANS | ...

(PARTIE IDIOMATIQUE) <-- (DESCRIPTION IDIOMATIQUE)(PARTIE IDIOMATIQUE)

(DESCRIPTION IDIOMATIQUE) <-- & (CHAINE)(DESCRIPTEUR)

4.5. Catégories grammaticales

Les notations utilisées pour représenter les catégories grammaticales des mots du dictionnaire sont les suivantes :

Na <-- Substantif
Ni <-- Substantif pluriel irrégulier
Ai <-- Adjectif
Aa <-- Adverbe
Ae <-- Article
Ca <-- Conjonction
Ne <-- Pronom
B <-- Abbréviation
Ce <-- conjonction de subordination
P <-- Préposition
Va <-- Verbe infinitif
Ve <-- Verbe irrégulier passé
Vi <-- Verbe irrégulier participe passé
V <-- Verbe conditionnel

Vê <-- Verbe irrégulier présent (accords spéciaux)

4.6 Exemple

Voici, à titre d'exemple, quelques lignes tirées du dictionnaire :

```
blow   £Na=FrapO
blow   £Va=BlovI[I/SpirEgI[I/SonI[I/SonIghI
blow   &blow up=EksplodI[I
blown  £Vi>blow
blunt  £Ai=MalAkrA/AbruptA
board  £Na=TabulO/KartonO
board  £Va=SurIrI/En[+N+IghI
board  &boarding school=EdukPensionO
board  &on board=BordE De
boat   £Na=BoatO/ShipO
...
```

Chapitre 5 : Choix du processus de traduction

5.1. Analyse des répertoires

Nous avons analysé les répertoires et avons recensé les types de phrases ou les types de constructions grammaticales qu'on y rencontrait.

Après une analyse approfondie, on s'est rendu compte que la syntaxe de ces répertoires était très complexe; en effet, beaucoup de phrases sont de style télégraphique (beaucoup de phrases n'ont pas de verbe; beaucoup sont du type: deux mots séparés par une virgule ("reading, while") etc...).

Cela signifie qu'ayant une phrase, il nous manque beaucoup d'informations essentielles pour en dégager sa structure, et donc pour la traduire. Ces informations se trouvent généralement dans ce qu'on appelle le "contexte" de la phrase, c'est à dire des phrases de niveau inférieur (cfr.1.2.) ayant un lien sémantique direct avec cette phrase (au maximum une phrase par niveau peut appartenir au contexte d'une phrase).

Le problème est que nous avons constaté qu'il n'était pas possible de trouver des régles permettant de lier les éléments d'une phrase aux éléments de son contexte.

Exemple:

On trouve dans le répertoire, au niveau 4, la phrase "from cold", qui suit la phrase de niveau 2 "air". Au niveau 4, "cold" se rapporte au nom "air" du niveau 2, et est donc un adjectif.

Cependant, il existe des phrases contenant également "cold", dans lesquelles "cold" n'est plus un adjectif mais représente le nom "cold" (qui signifie "un froid"). Le probléme est qu'on ne dispose d'aucun moyen, lors de l'analyse de "cold" pour savoir si "cold" est un adjectif ou un nom, d'autant plus que c'est primordial pour la traduction en Esperanto.; en effet, on a une terminaison différente suivant que c'est un nom (-O) ou un adjectif (-A).

Il aurait donc pratiquement fallu réécrire une grammaire de l'anglais, et ce n'était évidemment pas envisageable dans le cadre de ce mémoire...

On en a conclu qu'écrire une grammaire pour représenter ces phénomènes nécessitait un nombre de règles beaucoup trop important, et qu'un système basé sur une telle grammaire risquait de donner une traduction très approximative, qui ne serait pas correcte dans tous les cas; en outre, dans un tel système, la tâche du traducteur humain serait tellement importante (il aurait constamment à intervenir pour faire des corrections...) qu'il préfèrerait vraisemblablement traduire les répertoires lui-même.

C'est cette dernière conclusion qui nous a donné l'idée de faire directement traduire les répertoires en Esperanto, par un traducteur humain, avec l'aide de différents outils informatisés.

Même si cette solution pouvait, à première vue, paraître grotesque, elle nous est apparue comme la meilleure dans le cadre de notre travail.

5.2. Avantages de la traduction "manuelle"

Le gros avantage de cette méthode est que la traduction en Esperanto se fera sous forme de phrases "normales", vu qu'elle sera réalisée par un traducteur humain connaissant l'Esperanto et ayant à sa disposition tout le contexte de la phrase à traduire.

Etant donné que les phrases Esperanto qu'on obtiendra à l'issue de cette première phase seront complètes, les difficultés rencontrées lors de l'analyse des phrases anglaises du répertoire ne devraient plus se poser lors de la traduction en langue-cible (notamment l'analyse syntaxique des phrases Esperanto devrait s'en trouver grandement facilitée). En effet, toute l'information nécessaire pour traduire une phrase Esperanto en langue-cible devrait ainsi se trouver dans la phrase Esperanto elle-même.

Par exemple, si une phrase anglaise du répertoire contient un verbe mais pas le sujet de ce verbe, celui-ci figurera quand-même dans la traduction Esperanto (éventuellement marquée pour indiquer qu'il ne doit pas figurer dans la traduction en langue-cible, si l'on désire une traduction vraiment fidèle au texte anglais).

En outre, le mot Esperanto porteur du contenu sémantique du mot-clé anglais sera marqué afin de pouvoir le ré-identifier dans la version en langue-cible.

Après cette première phase de traduction, on disposerait donc de phrases

Esperanto complètes pour lesquelles il devrait normalement être possible d'écrire une grammaire cohérente, et surtout, un système de traduction (semi-)automatique en une langue-cible.

Nous avons donc opté pour cette technique de traduction, et nous avons construit plusieurs outils automatisés qui aideront le traducteur et simplifieront sa tâche (et qui seront détaillés plu tard [cfr. Chap.6]) :

- un dictionnaire automatisé Anglais-Esperanto.

- des primitives d'accès à ce dictionnaire; le traducteur tape le mot anglais dont il veut la traduction, et toutes les rubriques du dictionnaire concernant ce mot sont affichées à l'écran (les catégories grammaticales, les traductions, les expressions idiomatiques,...).

- une primitive qui affiche à l'écran la phrase à traduire, son contexte, et sa traduction si elle en a déja une; dans ce cas, cette traduction peut être modifiée par le traducteur humain.

Nous avons remarqué que certaines phrases revenaient un nombre élevé de fois dans les répertoires, notamment des expressions du type "reading, while", ou des expressions concernant l'heure, etc.

Nous avons donc pensé à faire traduire ces phrases une seule fois par un traducteur humain, et à recopier la traduction partout où la phrase apparaît. Cela évite surtout au traducteur de devoir retaper la même traduction un grand nombre de fois. Le programme permet également au traducteur de modifier une de ces traductions lors de la traduction réelle.

Cette pré-traduction se réalise en trois phases qui ont donné lieu à trois programmes:

-un programme qui compte le nombre d'occurences de toute phrase présente dans le répertoire, et qui sélectionne celles qui y sont présentes un nombre minimal donné de fois (fixé à dix pour des raisons pratiques).

- un programme qui permet de traduire les phrases sélectionnées, ainsi qu'un certain nombre de phrases simples (celles concernant l'heure), avant la session de traduction réelle du répertoire.

- enfin, un programme qui parcourt tout le répertoire, et chaque fois qu'il rencontre une des phrases sélectionnées, la recopie avec sa traduction.

Nous allons d'abord rappeler en quoi consiste le processus de traduction normal pour une approche interlinguale :

1. Analyse du texte-source.
2. Représentation de cette structure en termes de l'interlangage.
3. Conversion de cette représentation dans l'interlangage.
4. Vérification de la correction grammaticale du texte obtenu.

Ces quatre étapes ont été, dans notre version du système de traduction, remplacées par la phase de traduction "à la main" décrite ci-dessus (et nous avons donc abandonné le processus de traduction proposé initialement par JP Hogne, et décrit en 3.4).

Les phrases qui étaient présentes au moins dix fois dans le répertoire, et qui ont pu être traduites avant la phase de traduction elle-même étaient du type suivant :

-backward
-downward
-forward
- etc (mots se terminant par "ward")
-agg.
-amel.
-chiffre a.m.
-chiffre p.m.
-chiffre k.
-while suivi d'un radical verbal en "ing" (ex: "while writing",...)
-un nom seul au singulier (sauf les mots sur le temps); ex: "forehead", "occiput", ...
-un nom seul au pluriel; ex : "sides", "temples", ...
-un nom concernant le temps seul; ex : "afternoon", "daytime", ...

Pour un chapitre d'environ 5000 K., une centaine de phrases ont été sélectionnées, et plus de la moitié d'entre elles ont pu être traduites une fois pour toutes (pour les autres, il nous manquait certaines informations, notamment pour effectuer les accords).

Cette traduction Esperanto a été entièrement réalisée avec succés, pour un gros chapitre du KENT.

Il reste donc maintenant à réaliser les deux étapes suivantes:

5. Analyse du texte en interlangage pour obtenir une description intermédiaire de la structure.

6. Génération du texte en langue-cible à partir de cette description.

Nous allons, avant de définir en quoi consiste exactement cette traduction Esperanto-langue-cible, décrire plus en profondeur comment a été réalisé la traduction Esperanto, et nous allons spécifier les différents outils créés.

Chapitre 6 : Programmes utilitaires

6.1. Programme de sélection des phrases qui apparaissent au moins dix fois dans le répertoire

6.1.1. Présentation

Le programme reçoit en entrée un fichier contenant un répertoire SYNTHESIS, et il crée un fichier qui contient toutes les phrases qui apparaissent au moins dix fois dans ce répertoire. Le fichier SYNTHESIS en entrée est supposé syntaxiquement correct.

6.1.2. Stratégie de l'algorithme principal

La première partie du programme consiste à compter le nombre d'occurences de chaque phrase du fichier d'entrée.

Pour cela, on va créer un arbre binaire, qui va contenir un noeud pour chaque phrase distincte du fichier d'entrée. Chaque noeud contient une variable qui pointe sur le texte de la phrase, une variable qui contient le nombre d'occurences de la phrase, une variable qui pointe sur le noeud suivant de gauche, et une variable qui pointe sur le noeud suivant de droite.

L'arbre est construit de telle manière que, en n'importe quel noeud, le sous-arbre de gauche ne contienne que des phrases de valeur inférieure (par ordre alphabétique) à celle de la phrase du noeud, et que le sous-arbre de droite ne contienne que celles de valeur supérieure.

Pour savoir si une phrase appartient déja à l'arbre, on part de la racine, et on compare la phrase à celle mémorisée dans ce noeud-là. Si elles sont égales, on augmente le nombre d'occurences de ce noeud de un; si la phrase a une valeur inférieure, on la compare au noeud suivant de gauche, et dans le cas contraire, on la compare au noeud suivant de droite. S'il n'existe aucun noeud dans la direction désirée, cela signifie que la phrase n'existe pas dans l'arbre, et on crée un nouveau noeud la contenant, à l'endroit de ce noeud manquant.

Quand on a parcouru tout le fichier d'entrée, on écrit dans un fichier toutes

les phrases dont le nombre d'occurences est supérieur ou égal à dix.

6.1.3. Elaboration du programme

a) Variables globales

RepIn	: fichier d'entrée
NomRepIn	: nom du fichier SYNTHESIS en entrée
RepPhra	: fichier résultat contenant les phrases sélectionnées
NomRepPhra	: nom du fichier résultat

b) Procédures utilitaires

◇ Ouvrirfichier () : ouvre RepIn en lecture, et RepPhra en écriture.

◇ transf_en_char (ptPhrasein,ptresult) : transforme la CdC ptPhrasein, en la chaîne de caractères ptresult.

◇ reverse (s) : inverse la chaîne de caractères s.

◇ StopErreur (message) : provoque l'arrêt du programme, après fermeture de tous les fichiers ouverts et affichage de son argument.

c) Structure C d'un noeud de l'arbre

```
struct node {
    char *Phrase;              /* texte de la phrase */
    int  occurence;            /* nombre d'occurences */
    struc node *Phragauche;    /* pointeur vers le sous-arbre de gauche */
    struc node *Phradroite;    /* pointeur vers le sous-arbre de droite */
        }
```

6.1.4. Algorithme de la procédure arbre

Une phrase est présentée au plus haut niveau de l'arbre; à chaque niveau, on compare cette phrase à celle contenue dans le noeud , puis on descend d'un étage en empruntant soit le sous-arbre de gauche, soit celui de droite, et on recommence à l'aide d'un appel récursif à la fonction arbre.

Soit la phrase appartient déja à l'arbre binaire (et on incrémente alors le compteur correspondant), soit on doit créer un nouveau noeud et l'ajouter à l'arbre (si on rencontre un compteur nul); ce nouveau noeud est mémorisé par la fonction standard "calloc", et la fonction "arbre" transmet une variable qui pointe sur ce noeud.

```
fonction arbre(r,p)

si (r = NULL) : créer un nouveau noeud r;
        mémoriser r;
        copier la phrase p dans Phrase de r;
        mettre occurence de r à un;
        faire pointer Phragauche et Phradroite vers rien;

sinon :
  si (phrase p = Phrase de r) :
        incrémenter occurence de r;

  sinon :
        si (valeur de p est < valeur de phrase de r)
            arbre(r->Phragauche,p);

        sinon :
            arbre(r->Phradroite,p);

renvoyer r;
```

6.1.5. Spécifications des procédures auxiliaires

◇ lireelt (ptPhraseIn) :

fonction qui lit la phrase ptPhraseIn dans le fichier RepIn. Si elle a bien trouvé une phrase, elle prend la valeur "0", sinon, elle prend la valeur "1". Elle positionne RepIn sur le premier caractère qui suit la séquence lue.

◇ imprimerarbre (r) :

fonction qui écrit dans le fichier RepPhra, les phrases de l'arbre qui appartiennent à un noeud dont occurence est >= à dix.

◇ Ecrirephrase (p,o) :

fonction qui écrit la phrase p, et son nombre d'occurences o, dans le fichier RepPhra.

6.2. Programme de traduction des phrases sélectionnées

6.2.1. Présentation

Le programme reçoit en entrée le fichier contenant les phrases sélectionnées (résultat du programme 6.1), et il crée un nouveau fichier qui contient toutes les phrases suivies de leur traduction.

6.2.2. Procédures utilitaires

transf_en_char : ->[cfr. 6.1.3 b)]

6.2.3. Elaboration du programme

a) Variables globales

Foccur : fichier d'entrée.
Ftrad : fichier résultat.
Dict : fichier contenant le dictionnaire anglais-Esperanto.

b) Conditions invariantes au point d'itération

-la position courante dans Foccur est sur le premier caractère qui n'a pas été traité.

-le fichier Ftrad contient les phrases de Foccur suivies de leur traduction, allant du début à la position courante non comprise.

c) Algorithme principal

◇ *Initialisation*:
rewind (Foccur); /* repositionne le fichier au début */

◇ *Itération*:
recopier la phrase courante de Foccur dans la variable Phrase;
si (dernier caractère lu = EOF) :
sortir de l'itération;

afficher la phrase à l'écran; /* AfficherPhrase(&pt) */
traduire la phrase; /* Traduire(&pt,&cxt) */
prendre la traduction;
copier la traduction dans la variable traduc;
écrire Phrase dans Ftrad;
écrire traduc dans Ftrad;

6.2.4. Procédures auxiliaires

◇ AfficherPhrase (PhraTra) :
fonction qui affiche à l'écran la phrase contenue dans la PhraseTrad PhraTra;

◇ AfficherContexte (PhraTra,ContexPhra) :
fonction qui affiche à l'écran le contexte de la phrase contenue dans PhraTra (c'est à dire les phrases contenues dans ContexPhra); elle affiche,aussi la traduction de la phrase, si celle-ci existe.

◇ Traduire (ptPhraTra, ptContexPhra) :
fonction qui fait traduire par l'utilisateur la phrase contenue dans ptPhraTra; celui-ci peut demander d'accéder au dictionnaire pour connaître les traductions de différents mots. Elle fait afficher la phrase, son contexte, sa traduction éventuelle, et demande à l'utilisateur d'introduire la traduction au clavier (ou de modifier la traduction existante, s'il le désire).

□ Primitives d'accès au dictionnaire :

◇ CherchеMotsDico :
fonction qui demande à l'utilisateur d'introduire un mot, et qui cherche ce mot dans le dictionnaire; s'il s'y trouve, elle affiche toutes les rubriques du dictionnaire concernant ce mot.

◇ LireMotDict (MotDict, DescrMotDict) :
fonction qui lit la ligne courante du fichier dictionnaire Dict, et qui met le mot lu dans MotDict, et sa description (catégorie grammaticale, traductions,etc...) dans DescrMotDict.

◇ AfficherMot (MotDict,DescrMotDict) :
fonction qui affiche à l'écran le mot contenu dans Motdict, et la description de ce mot, contenue dans DescrMotDict, sur une même ligne.

6.3. Programme qui recopie dans le répertoire les phrases dèja traduites ainsi que celles qui concernent l'heure

6.3.1. Présentation

Le programme reçoit en entrée un fichier contenant un répertoire SYNTHESIS, et le fichier contenant les phrases sélectionnées et leur traduction; il crée un autre fichier répertoire qui a le même contenu que le premier, excepté pour les phrases qui appartiennent au second fichier, pour lesquelles il recopie leur traduction, et pour les phrases simples concernant l'heure qu'il traduit automatiquement.

Ces dernières phrases étant présentes un très grand nombre de fois dans le répertoire, et leur traduction étant assez simple, celle-ci peut être faite automatiquement, au fur et à mesure que l'on rencontre ces phrases; on évite ainsi un travail répétitif au traducteur humain.

En fait, les phrases concernées sont de trois types:

x a.m. se traduit en x atm.
x p.m. se traduit en x ptm.
x h. se traduit en x h.
(où x est un nombre quelconque)

6.3.2. Elaboration du programme

a) Variables globales

Ftrad : fichier contenant les phrases sélectionnées et leur traduction
RepIn : fichier répertoire en entrée
RepOut : fichier répertoire en sortie

b) Procédures utilitaires

InitPreSession () : ouvre le fichier Ftrad en lecture.

c) Conditions invariantes au point d'itération

- La position courante dans RepIn est sur le premier caractère qui n'a pas été traité.

- Le fichier RepOut contient la partie de RepIn allant du début jusqu'à la position courante, dans laquelle les phrases qui appartiennent au fichier Ftrad et les phrases concernant l'heure sont assorties de leur traduction.

d) Algorithme principal

```
prendre une PhraseTrad dans le répertoire (soit PhraTra);
si (il n'y a plus de PhraseTrad) : renvoyer '0';
sinon : prendre la phrase contenue dans PhraTra (soit Phrase);
        si (le premier caractère de Phrase est un chiffre)
           /* il s'agit d'une heure */
           TraduitHeure(PhraRep,traduc);
           si (le format est bien celui d'une heure) :
              mettre la traduction traduc dans PhraTra;

        sinon :
           repositionner Ftrad au début;
           chercher la phrase Phrase dans Ftrad;
           si (Phrase appartient à Ftrad) :
              prendre la traduction de la phrase dans Ftrad;
              mettre la traduction dans PhraTra;

        remettre PhraTra dans le fichier RepOut;
```

e) Procédures auxiliaires

◇ PrendrePhrase (p) :
fonction qui lit une phrase dans le fichier Ftrad, et la renvoie dans p.

◇ PrendreTraductionFich (t) :
fonction qui lit la traduction d'une phrase dans le fichier Ftrad, et la renvoie dans t.

◇ TraduitHeure (p,c) :
fonction qui reçoit la chaîne de caractères p, et qui vérifie si elle est bien d'un des trois types d'heure standarts. Dans ce cas, elle traduit p (conformément aux régles dictées en 3.1), met sa traduction dans c, et renvoie '0'; sinon, elle renvoie '1'.

6.4.Programme qui réalise la traduction anglais-Esperanto

6.4.1 Présentation

Le programme reçoit en entrée un fichier SYNTHESIS, et il crée un nouveau fichier qui contient éventuellement les traductions des phrases du fichier.
(La traduction est réalisée par l'utilisateur, phrase par phrase).

6.4.2. Elaboration du programme

a) Variables globales

Dict	: fichier dictionnaire Anglais-Esperanto
NomDict	: nom du fichier dictionnaire

b) Procédures utilitaires

OuvrirDico ()	: ouvre Dict en lecture.

FermerDico () : ferme Dict.
PoursuivreSession() : renvoie TRUE si l'utilisateur veut poursuivre la session de traduction, FALSE sinon.

c) Procédure Session

◇ Conditions au point d'itération :

- la position courante est sur le premier caractère qui n'a pas été traité.

- le fichier RepOut contient la partie de RepIn allant du début jusqu'à la position courante non inclue, dans laquelle les phrases sont éventuellement assorties de leur traduction.

◇ Algorithme :

```
prendre une PhraseTrad dans RepIn (soit PhraTra);
si (il n'y a plus de PhraseTrad) : renvoyer '0';
sinon : si (PoursuivreSession() = FALSE ) :
            remettre PhraTra dans RepOut;
            quitter l'itération;

        sinon :
            prendre la phrase contenue dans PhraTra (soit Phra);
            AfficherPhrase(PhraTra);
            AfficherContextePhrase(PhraTra,ContexPhra);
            si (Phra n'a pas de traduction) :
                Traduire(PhraTra,ContexPhra);

            remettre PhraTra dans RepOut;
```

◇ Procédures auxiliaires :

- AfficherPhrase, AfficherContexPhrase, Traduire : [cfr. 6.2.4].

6.5. Exemple tiré du répertoite traduit

&1rainy weather
PluvA VeterO:
mag-c.2:

&2rubbing amel.
FrotI :
dros.2nat-m.1:

&3scratching
GratI:

&4changes place,after
ShanghO De LokO Post :
cycl.2mez.2sars.1staph.1:

&4not amel.after
Ne Post GratI:
bov.1calc.1carb-an.1:

&3sleep,when going to
Kiam EkDormAs:
agn.1:

&3spots
LokOJ:
sil.1:

&3sudden
SubitA:
ph-ac.1:

N.B : &1 indique que la rubrique qui suit est de niveau 1 dans le répertoire

Partie III :

La traduction Esperanto-français

Chapitre 7 : Présentation générale

Nous disposons maintenant d'une version des répertoires "sous-titrée" en Esperanto, il reste donc à la sous-titrer en français -langue-cible qui a été choisie pour ce premier système de traduction.

7.1. Etapes de la traduction

Cette traduction de l'Esperanto en français va se dérouler en deux étapes.

- **Etape 1** : elle consiste à analyser la phrase Esperanto à traduire, d'après le dictionnaire Esperanto-langage-cible, et les régles grammaticales internes, afin d'obtenir une structure arborescente en Esperanto; il s'agit de ce qu'on appele couramment l'analyse syntaxique de la phrase.

Le mot porteur du contenu sémantique (et marqué dans la phrase Esperanto) est également marqué dans la structure obtenue.

De plus, une étape de vérification grammaticale doit être insérée à ce niveau-ci (basée sur des régles de correction).

- **Etape 2** : il s'agit de la conversion de la structure Esperanto en une structure en langue-cible, puis de la traduction des groupes de mots de cette structure.

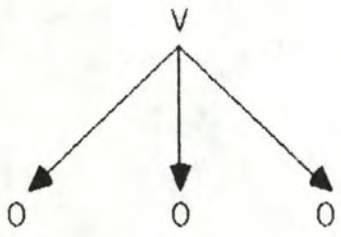
Le mot marqué dans la structure en langue-cible est projeté en tête de la phrase, une virgule est ajoutée après ce mot, et le reste de la phrase est ensuite retranscrit dans l'ordre correct; cet ordre est déterminé par l'exécution de certaines régles de réécriture (ou régles de métataxe).

7.2. Exemple

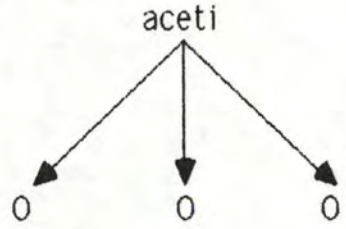
Soit la phrase anglaise "purchase, making" du répertoire.

Sa traduction en Esperanto est "acheti".

L'analyse syntaxique Esperanto va nous dire qu'il s'agit d'un verbe qui admet un sujet, un complément d'objet direct, et un complément d'objet indirect; la structure arborescente correspondant à un tel verbe est la suivante:



L'analyse de "acheti" va nous donner la structure arborescente suivante:



Après consultation du dictionnaire, on obtient "acheti : acheter / faire des achats".

On convertit la structure Esperanto de "acheti" en structure en français :

Enfin, le mot souligné est placé en tête de la phrase, et le reste de la phrase est retranscrit ensuite dans l'ordre correct...

On obtient alors la phrase française : "achats, faire des".

7.3. Travail réalisé

Dans le cadre de ce mémoire, nous avons réussi à encoder un dictionnaire Esperanto-français [cfr. Chap. 8], à écrire une grammaire Esperanto et à la formaliser sous forme d'A.T.N. [cfr. Chap. 10 et 11], et à écrire l'analyseur syntaxique des phrases Esperanto, c'est à dire le programme qui réalise l'étape 1 de la traduction [cfr. Chap. 12].

Nous n'avons malheureusement pas eu le temps d'implémenter le programme de génération du texte en français à partir de la description obtenue par l'analyseur syntaxique, mais une bonne partie du travail préparatoire a été réalisé [cfr. Chap. 13].

Chapitre 8 : Le dictionnaire Esperanto-français

8.1. Introduction

Etant donné que pour l'analyse syntaxique, et surtout pour la réécriture en français, nous avons besoin de diverses informations concernant chaque terme Esperanto, nous avons été obligés de définir une syntaxe beaucoup plus formelle pour le dictionnaire Esperanto-français que celle employée pour l'anglais-Esperanto.

Nous avons d'abord recensé tous les mots Esperanto figurant dans les répertoires déjà traduits, et nous avons recherché la racine sémantique de chacun de ces mots.

Nous avons alors écrit le dictionnaire à partir de ces racines.

8.2. Présentation

Le dictionnaire est une suite de rubriques qui concernent chacune une racine Esperanto.

Chaque rubrique reprend tous les mots Esperanto et/ou toutes les différentes catégories grammaticales qui peuvent être construites à partir de cette racine, accompagnés de leurs traductions.

Un mot Esperanto est construit en collant à une racine sémantique un préfixe et/ou des suffixes. Certains de ces affixes modifient la catégorie grammaticale du mot ainsi construit [cfr. 3.2.].

Par exemple : -O : substantif , -A : adjectif, -E : adverbe, -I : verbe à l'infinitif, -As : présent, -Os : futur, -Us : conditionnel, etc (cfr. 2.2).

D'autres modifient son sens : -Ebl : suffixe fonctionnel qui correspond à '-able' en français, -Il : instrument, Mal- : contraire, etc.

Tous ces affixes sont repris, pour tout mot, sur une ligne distincte.

Cependant, une racine peut elle-même être un mot Esperanto, qui peut posséder différentes catégories grammaticales; dans ce cas, chaque catégorie grammaticale est reprise sur une ligne différente, avec sa traduction.

En outre, pour chaque mot Esperanto dérivé d'une racine, on écrit un certain nombre d'informations : valence du verbe, prépositions utilisées avec tel verbe, etc.

Pour les mots composés, on inscrit dans le dictionnaire, un caractère particulier (O) entre la racine et le mot.

8.3. Exemple de rubrique

§Abund
[X*A = abondant] /* adjectif "AbundA" qui signifie "abondant" */

[X*E = abondamment] /* adverbe "AbundE" qui signifie "abondamment" */

[X*O = abondance] /* substantif "AbundO" qui signifie "abondance" */

[X*I <V1,V3Je> = abonder] /* verbe "AbundI" qui admet un sujet et un complément d'objet indirect introduit par la préposition "Je", et qui signifie "abonder" */

[X*Ec*O = abondance] /* substantif "MalAbundEcO" qui signifie "abondance" */

[Mal*X*A = insuffisant] /* adjectif "MalAbundA" qui signifie "insuffisant" */

[Mal*X*Ec*O = pénurie] /* substantif "MalAbundEcO" qui signifie "pénurie" */

[Super*X*I <V1,V3Je> = surabonder] /* verbe "SuperAbundI", qui admet un sujet et un complément d'objet indirect introduit par la préposition "Je", et qui signifie "surabonder" */

[X*O*Korn*O = corne d'abondance] /* mot composé "AbundOKornO" qui signifie "corne d'abondance" */

8.4. Conventions

Le tableau 8.1 reprend tous les caractères utilisés pour l'encodage du dictionnaire Esperanto-français.

Pour indiquer...	Caractère
un début de rubrique	&
un début de catégorie de mot	[
une fin de catégorie de mot	]
la place de la racine dans le mot	x
une nouvelle partie du mot (suffixe ou racine)	*
une information sur la valence du verbe	<...,...>
la séparation, dans un mot composé, entre la racine et la deuxième partie du mot	0
le début de la traduction française	=

tableau 8.1

8.5. Règles syntaxiques

(DICO) <-- (RUBRIQUE) | (RUBRIQUE) (DICO)

(RUBRIQUE) <-- (DEBUT) (DESCRIPTION RUBRIQUE)

(DEBUT) <-- **&**(CHAINE DE CARACTERES)

(DESCRIPTION RUBRIQUE)<-- (DESCRIPTION MOT) | (DESCRIPTION MOT)(DESCRIPTION RUBRIQUE)

(DESCRIPTION MOT) <--**[**(PREFIXE)$^{0..1}$ * X * 0$^{0..1}$ * (SUFFIXE)$^{0..\infty}$ **<**(VALENCE)**>**$^{0..1}$
= (TRADUCTION)$^{1..\infty}$ **]** | **[** (CATEGORIE) = (TRADUCTION)$^{1..\infty}$ **]**

(PREFIXE) <-- Mal | Mis | ...

(SUFFIXE) <-- Ec | A | O | E | As | ...

(VALENCE) <--V1| V1,V2 | V1,V3 **(**(PREP)**)**$^{0..1}$ | V1,V2,V3 **(**(PREP)**)**$^{0..1}$ | V2

(PREP)<-- Je ...

(TRADUCTION) <-- (CHAINE DE CARACTERES)

(CATEGORIE) <-- P | Ne | B | ...

N.B : Les notations utilisées pour les catégories grammaticales sont les mêmes que celles utilisées pour le dictionnaire anglais-Esperanto [cfr. 3.1.5].

Chapitre 9 : Choix du modèle syntaxique

9.1. Petit historique

De tout temps, les hommes se sont interrogés sur la nature de leur langage et pourtant, la linguistique, en tant que discipline scientifique, est relativement jeune.

Elle n'a véritablement acquis ses lettres de noblesse qu'avec le linguiste suisse Ferdinand de Saussure (1857-1913) et les "Cours de linguistique Générale" qu'il dispensa, à la fin de sa vie, à l'université de Genève.

Avant lui, mises à part les traditionnelles grammaires normatives, ceux qui étudiaient le langage s'étaient bornés à des recherches historiques ou à des essais d'apparentement des langues, parfois tout à fait fantaisistes (allant jusqu'à lier nature de la langue et nature des cheveux de ses locuteurs !).

Saussure a été le premier à se dégager de l'historicisme ambiant et à observer la langue pour elle-même, en dehors de tout a priori génétique, racial, religieux ou autre.

Il a ainsi pu établir que toute langue est un système bien défini, une structure qui n'obéit qu'à ses propres lois. Saussure a alors donné naissance à toute une école structuraliste, qui a largement débordé des limites de la linguistique pour s'étendre à presque toutes les sciences humaines (par exemple, Cl. Lévi-Strauss en anthropologie).

Si les disciples de Saussure, en linguistique comme ailleurs, se sont divisés en courants largements divergents, Lucien Tesnière, dont il sera question plus loin, est probablement l'un de ses continuateurs les plus fidèles et l'un de ceux qui ont su le mieux approfondir les grandes intuitions de l'illustre maître genèvois.

Le structuralisme, en effet, s'est vite transformé en un "puzzle" de théories et on a même pu observer une véritable cassure entre les linguistes européens, plus "traditionnalistes", et les linguistes américains, qui, confrontés aux langues amérindiennes, structurellement totalement différentes des langues connues jusqu'alors, ont été forcés de remettre en question bien des données que l'on tenait pour assurées.

C'est ce courant américain du structuralisme qui a formé Noam Chomsky et dont il s'est nettement démarqué au fil de ses recherches, soulignant avec de plus en plus de vigueur les lacunes des théories élaborées par ses prédécesseurs (notamment au niveau de l'analyse sémantique) et allant curieusement se retremper aux sources des grammaires "logiques" et normatives.

C'est pourquoi, contrairement à certains auteurs (comme Leroy, par exemple) qui l'y maintiennent, nous estimons pouvoir opposer la grammaire transformationnelle de Chomsky au courant structuraliste. De même que nous en sommes venus à douter de la capacité du modèle linguistique de Chomsky à pallier les lacunes qu'il a soulignées chez d'autres.

9.2. La grammaire Générative et Transformationnelle

Nous ne nous attarderons pas à analyser ce modèle trop en détails [CHOMSKY]. Chomsky étant le principal inspirateur des travaux de linguistique appliquée à la traduction automatique, sa méthode d'analyse syntaxique et les notions concomittantes de structure de surface et de structure profonde qu'il a développées sont en effet familiers aux informaticiens.

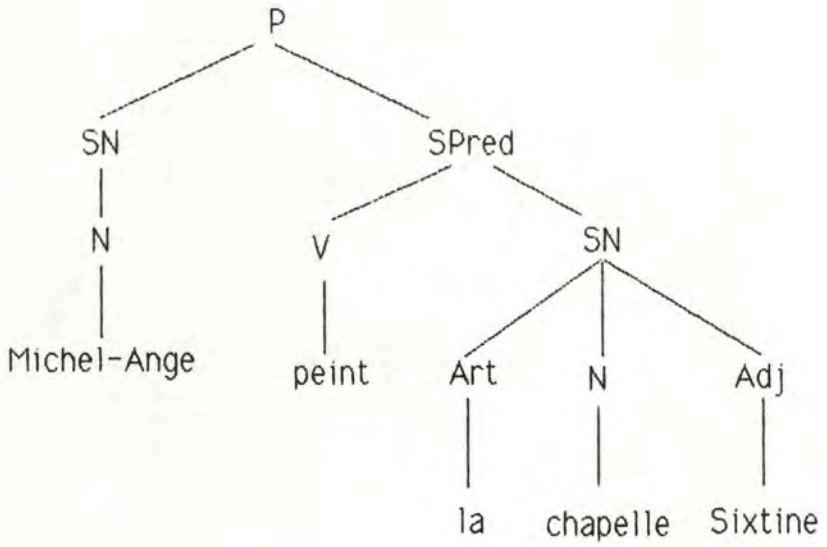
Nous nous contenterons de redéfinir brièvement ces notions pour mémoire.

9.2.1. Le modèle d'analyse syntaxique

Le modèle d'analyse syntaxique en lui-même ne se démarque pas encore du courant structuraliste. Chomsky propose un découpage de la phrase en syntagmes, ceux-ci étant à leur tour découpés en éléments constitutifs. Ces syntagmes et ces éléments sont exprimés en termes de catégories grammaticales (noms, verbes, adjectifs...). [SEMANT]

Schématisée sous forme d'arborescence, l'analyse donne ceci :

(Michel-Ange peint la chapelle Sixtine)

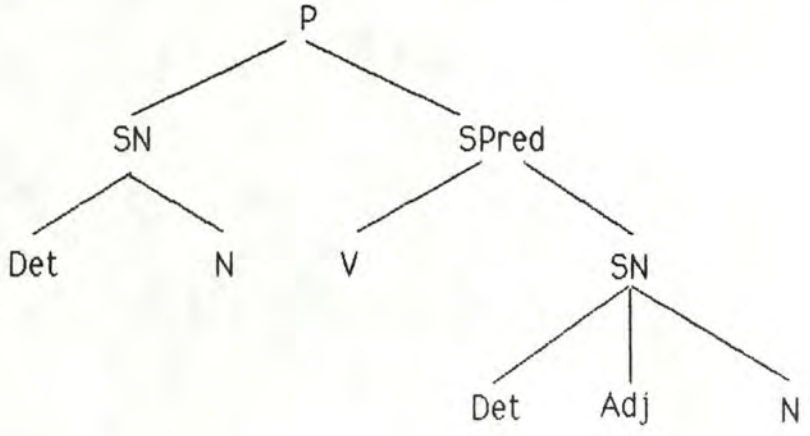


9.2.2. La notion de transformation

L'analyse syntaxique d'un échantillon de phrases permet de dégager toute une série d'arbres virtuels. Cependant, on se rend vite compte qu'à un même arbre virtuel peuvent se superposer plusieurs phrases qui, si elles ont la même structure, n'ont pas du tout la même interprétation sémantique. Par exemple, à l'arborescence dessinée ci-dessus peut également correspondre une phrase comme : Cézanne peint une nature morte.

C'est pour dissiper ce genre d'ambiguités que Chomsky introduit dans sa théorie la notion de **transformation**. Deux phrases sont différentes si elles ne peuvent subir les mêmes transformations et rester grammaticales. (Une phrase 'grammaticale' étant un énoncé accepté comme 'correct' par un locuteur natif).

Par exemple, les phrases 'un peintre a dessiné ce beau paysage' et 'le train a roulé deux longues heures' peuvent être représentées par le même arbre :



mais elles ne peuvent pas subir toutes deux la transformation passive. Si 'ce beau paysage a été dessiné par un peintre' est grammaticale, 'deux longues heures ont été roulées par le train' ne l'est pas.

9.2.3. Structure de surface et structure profonde

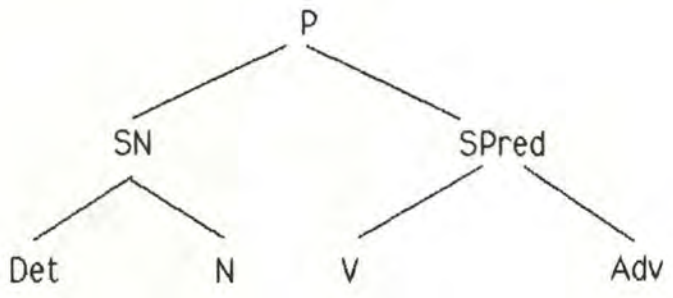
Si les deux phrases citées ci-dessus ne peuvent subir la même transformation passive, c'est qu'elles ont été générées par des structures profondes différentes.

La grammaire de Chomsky postule en effet que sous la structure de surface de l'énoncé (celle que nous entendons ou lisons) se cache(nt) une (plusieurs) structure(s) profonde(s) -proportionnellement à la complexité de l'énoncé.

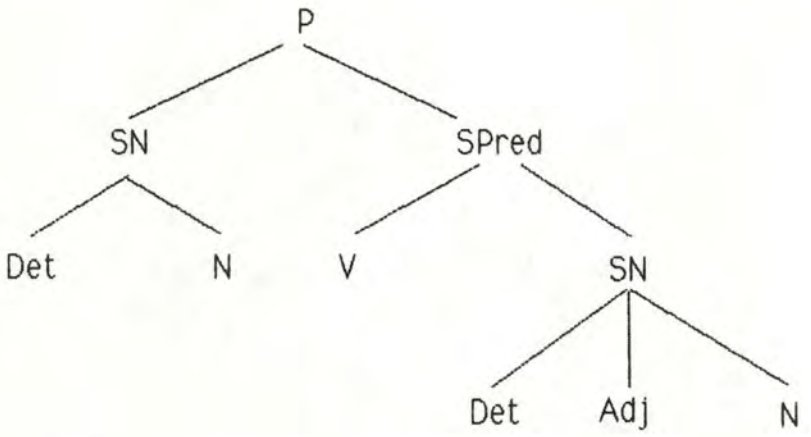
A partir de cette structure profonde, -phrase-noyau la plus simple possible, du type SN-SV-SN - on peut générer les structures de surface en passant par diverses transformations (nominale, passive, relative...).

La même structure profonde peut générer des structures de surface extrêmement différentes selon les transformations que le locuteur 'choisit' d'y appliquer. Ceci explique que nous puissions toujours produire des énoncés 'inédits'.

Pour en revenir aux exemples précédents, la phrase 'le train a roulé deux longues heures' correspond à une structure profonde :



et non à :



9.2.4. Critiques

Nous avons affirmé plus haut que nous ne considérons plus Chomsky comme un structuraliste. C'est autant pour sa conception de l'analyse et ses méthodes de travail que pour les innovations qu'il a introduites en linguistique.

En effet, et curieusement, Chomsky renie ses prédécesseurs et la majorité des progrès accomplis depuis Saussure, parce qu'il découvre des lacunes dans leurs modèles linguistiques, mais c'est au XVIIe siècle qu'il va chercher son inspiration, dans la très cartésienne "Grammaire Générale et Raisonnée de Port-Royal", ouvrage dont les auteurs soutiennent que l'on peut parvenir à apprendre une langue par la logique, la seule force du raisonnement.

Nous craignons que peu de pédagogues du XXe siècle partagent cette opinion...

La théorie de Chomsky, il est vrai, est tout imprégnée de cette volonté de respecter les données de la logique formelle. Cela se traduit notamment dans ses arborescences, dans lesquelles la première division de la phrase est la traditionnelle coupure entre sujet et prédicat. Coupure directement inspirée des méthodes d'analyse de la logique, mais qui, pour beaucoup de linguistes d'aujourd'hui, ne se justifie pas en grammaire.

Nous verrons plus loin à quelle conclusion Tesnière est arrivé à ce niveau, en observant, dans de nombreuses langues, le rôle du sujet par rapport au reste de la phrase, et ce, sans a-priori philosophique.

De même, il est permis de s'étonner d'une certaine évolution de Chomsky qui, dans "Syntactic Structures" affirme que la grammaire doit rester indépendante de toute sémantique, alors que c'est bien à partir de différences d'interprétation sémantique que se crée tout le système des transformations, des structures profondes...

Lui-même, d'ailleurs, essaiera plus tard de se servir de ce système comme base d'une nouvelle sémantique. L'absence de cet aspect dans les modèles structuralistes de ses prédécesseurs (notamment les distributionalistes) étant le principal reproche qu'il leur adressait.

A notre tour, nous pourrions quelque peu critiquer ses méthodes de travail. Chomsky veut bâtir une grammaire de l'anglais, mais n'analyse que ses propres productions d'énoncés , se fiant à son intuition de locuteur natif pour juger de la (non-)grammaticalité de ces mêmes énoncés. Démarche qui risque bien de faire dériver ses analyses vers une grammaire-de-l'anglais-parlé-par-Chomsky...

Enfin, lorsqu'on l'attaque sur l'absence dans son modèle de toute notion de fonction remplie dans la phrase par les syntagmes qu'il délimite -et ici, il marque un recul par rapport aux grammaires traditionnelles, aussi confuses soient-elles - , Chomsky se réfugie de nouveau derrière l'intuition du sujet parlant. Donnée difficile à intégrer dans un programme informatique, s'il en est.

9.3. La grammaire de dépendance

9.3.1. Introduction

Contrairement à Chomsky, Tesnière n'est pas un théoricien. C'est un professeur de langues. A l'origine de ses recherches, il y a surtout le souci de découvrir une démarche d'analyse grammaticale qui serait la même pour le plus grand nombre de langues possible. Démarche qui servirait aux étudiants de "pont" entre leur langue maternelle et la (les) langue(s) qu'ils souhaitent apprendre.

Et, comme nous l'avons dit précédemment, pour ce faire, Tesnière a observé le fonctionnement de toutes les langues qu'il connaissait en termes de comportement de la structure, de "l'intérieur"-même de la langue et non, comme l'ont fait Chomsky et beaucoup d'autres, en tentant de lui surimposer un moule.

Ceci débouche sur des vues tout-à-fait novatrices.

Comment expliquer alors que si peu de chercheurs s'en soient inspirés?

Cela tient sans doute en partie à la personnalité du linguiste, qui ne souhaitait pas publier à grande échelle le résultat de ses recherches tant qu'elles ne le satisfaisaient pas complètement.

Mais c'est sans doute dû surtout au désintérêt total des français (du moins à cette époque) pour les problèmes de traitement du langage (automatique ou non). Les recherches d'un Chomsky, par exemple, aux Etats-Unis, ont été largement financées par l'Etat et l'armée, et diffusées dans toutes les universités.

Peut-être aussi le style de Chomsky a-t-il plus d'attrait pour les ingénieurs et les informaticiens ?

En tout cas, Tesnière a su ne pas déborder du cadre de la linguistique et ceci donne à sa "syntaxe" [TESNIERE] une cohérence qui fait défaut chez beaucoup de ses confrères.

9.3.2. Présentation

Exposer l'entièreté des innovations de Tesnière par rapport à l'analyse traditionnelle est un travail qui mériterait plusieurs mémoires à soi tout seul.

Donc, tout comme pour la grammaire transformationnelle, nous nous bornerons à présenter certaines notions, qui sont à la base de son système et de notre travail, à savoir :

1. la connexion
2. la hiérarchie des connexions
3. le stemma
4. la chaîne parlée et l'ordre structural
5. la valence du verbe, les actants et les circonstants
6. le nucléus

9.3.3. La notion de connexion

La notion de **connexion** est à la base de toute la théorie.

Tous les linguistes sont d'accord sur le fait que la phrase est "un ensemble organisé dont les éléments constituants sont les mots" [Tesnière].

A cela, Tesnière ajoute un autre élément : la connexion, qui relie ces mots entre eux et sans laquelle toute phrase ne serait qu'une simple suite d'éléments indépendants les uns des autres.

Cette connexion n'est (en général) pas matérialisée, mais elle est perçue par l'esprit, elle est le ciment qui assure la cohésion de la structure "phrase" et sa signification.

Par exemple : "Alfred parle" n'est pas composé de deux mais bien de trois éléments : la personne "Alfred", l'action de parler, et la connexion qui fait entrer ces deux mots, au départ indépendants, en relation.

Graphiquement, les connexions entre les mots sont représentées par des traits appelés "traits de connexion".

Exemple :

9.3.4. Rapports de dépendance

Non seulement les mots qui composent une phrase sont reliés par une connexion, mais cette connexion est hiérarchisée (connexion et hiérarchie sont des intuitions de la grammaire traditionnelle qui n'avaient jamais été formulées).

Les connexions établissent entre les mots des rapports de **dépendance** (d'où le nom de "dependency grammar" attribué à ce que Tesnière avait intitulé "syntaxe structurale").

Elles relient un terme supérieur (le régissant) à un autre inférieur (le subordonné).

Ainsi, dans "Alfred parle", le verbe "parle" régit-il le sujet "Alfred".

Tout mot peut, évidemment, être à la fois régissant et subordonné.

Dans "Mon ami parle", "ami" est gouverné par "parle" mais régit "mon" :

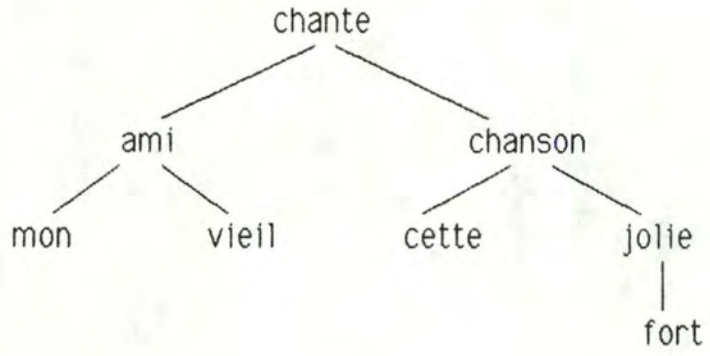
parle
|
ami
|
mon

9.3.5. Stemma

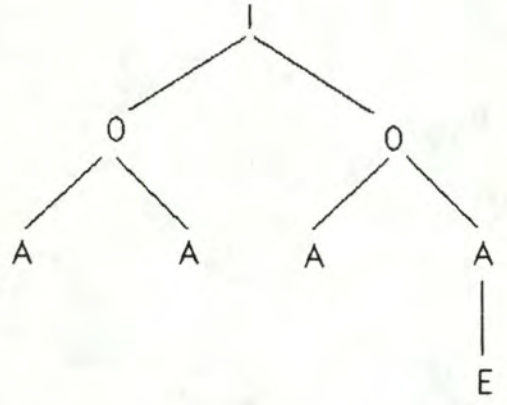
Pour représenter les relations entre constituants d'une phrase, Tesnière se sert lui aussi d'une arborescence, qu'il appelle "**stemma**" (du latin "arbre"), et qui se présente comme suit : chaque régissant représente un noeud de l'arbre et chaque trait représente la connexion; le verbe est le plus souvent le noeud supérieur, celui qui assure la cohésion de toute la phrase, qui réunit les divers éléments en un seul faisceau. (Mais en son absence, le noeud central peut être un nom, un adjectif ou un adverbe, par ordre de probabilité).

Tesnière définit l'arborescence comme "la représentation visuelle d'une notion abstraite qui n'est autre que le schème structural de la phrase"; notion sur laquelle nous reviendrons plus loin.

Exemple : la phrase "mon vieil ami chante cette fort jolie chanson." peut être représentée par le stemma réel suivant :



Bien sûr, on peut tirer des stemmas réels une série de stemmas virtuels qui seront représentés de cette manière :



stemma virtuel de l'exemple précédent

Les symboles qui figurent à chaque noeud correspondent, pour chaque catégorie grammaticale, aux terminaisons utilisées en Esperanto, à savoir:
I pour le verbe, O pour le nom, A pour l'adjectif et E pour l'adverbe (symboles réellemnt utilisés par Tesnière).

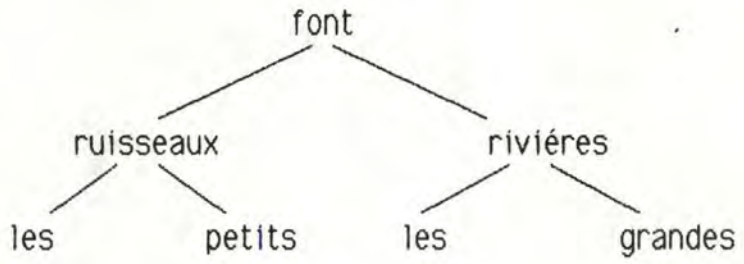
9.3.6. Ordre structural et ordre linéaire

Nous avons vu que le stemma "matérialise" les relations d'**ordre structural** entre les divers éléments d'une phrase et, en particulier, les multiples connexions inférieures que peut connaître un noeud.

Or, si les connexions peuvent avoir plusieurs dimensions (deux, dans la représentation stemmatique), la chaîne parlée (les phrases telles que nous les entendons ou prononçons), elle, n'en admet qu'une : l'action de parler (d'écrire) ne peut se dérouler que dans le temps et à sens unique.

De plus, quel que soit le nombre de ses connexions, un mot de la chaîne parlée ne peut se trouver en séquence avec plus de deux autres mots. Cela signifie qu'en pratique, il faut sacrifier certaines connexions au profit d'autres lorsqu'on transpose ces connexions structurales en séquences linéaires. Problème auquel l'accord grammatical fournit un remède efficace, en permettant de garder une trace de la connexion même si les séquences doivent être rompues. (Dans ce domaine, la rigueur grammaticale de l'Esperanto nous sera d'un grand secours pour le passage d'une langue à l'autre.)

Un exemple montrera clairement la différence entre l'ordre linéaire de la chaîne parlée et l'ordre structural :



Lors de la transformation en ordre linéaire, "ruisseaux" ne peut figurer en séquence qu'avec deux autres mots. D'abord, le verbe "font" qui est son régissant; ensuite, il faut choisir entre ses deux dépendants "les" et "petits". On choisit "petits" car il s'agit de son subordonné sémantiquement le plus important (c'est celui qui modifie le sens de "ruisseaux"), alors que "les" ne fait que redéfinir "ruisseaux" (marque du pluriel déja spécifiée par la terminaison "au**x**"). C'est du moins le procédé utilisé en français, car les connexions auxquelles on donne la préférence ne sont pas les mêmes dans toutes les langues.

9.3.7. Valence

Un autre grand mérite de Tesnière a été de mettre de l'ordre dans l'embrouillamini de compléments que comportent les grammaires traditionnelles, en introduisant la notion de **valence** du verbe et, partant, la différence essentielle entre compléments et circonstants. Ceci résout efficacement l'épineux problème de la transitivité.

Tesnière pose que chaque verbe peut régir un ou plusieurs actants. Un actant est une personne ou une chose qui participe au procès exprimé par le verbe.

Exemple : Dans "Antoine tombe", "tombe" est un verbe à un actant. L'action de tomber peut se réaliser sans l'intervention d'un autre actant.

Dans "Antoine frappe Bernard", le verbe a deux actants : celui qui frappe et celui qui reçoit les coups.

Dans "Antoine donne le livre à Bernard", nous sommes en présence de trois actants : Antoine, qui donne, le livre, qui est donné, et Bernard, qui reçoit.

On le voit, ces actants, s'ils sont tous des substantifs, ne remplissent pas le même rôle sémantique.

Le prime actant remplit le rôle du traditionnel sujet, le second actant est celui qui supporte l'action (ou objet), le tiers actant est celui au bénéfice ou au détriment duquel se fait l'action (cette définition du tiers actant peut être partois sémantiquement élargie).

La notion de valence du verbe (c'est à dire du nombre d'actants que le verbe peut accepter, exprimé par un chiffre de zéro à trois) permet d'évacuer définitivement les problèmes des verbes impersonnels, transitifs, doublement transitifs... (Ceci est extrêmement utile pour la rédaction des dictionnaires automatiques dont tout système de traduction a besoin).

Les trois actants constituent les compléments du verbe, tous les autres syntagmes qui lui sont connectés sont considérés comme des circonstants; ils expriment le temps, le lieu, la manière dont se déroule l'action.

A ce propos, nous avions critiqué, dans un chapitre précédent, le fait que Chomsky divise d'abord sa phrase en sujet et prédicat, donnant ainsi au sujet une importance que rien ne justifie, du moins en linguistique.

Pour Tesnière, le sujet est un complément comme les autres, ni plus ni moins important que les deux autres actants. Les arguments en faveur de la scission sujet / prédicat (et donc contre la conception du noeud verbal) "relèvent de la logique formelle a priori qui n'a rien à voir en linguistique".

L'observation des faits de langue conduit à de toutes autres conclusions que celles des grammaires traditionnelle et transformationnelle : "L'opposition du sujet au prédicat empêche (ainsi) de saisir l'équilibre structural de la phrase, puisqu'elle conduit à isoler comme sujet un des actants, à l'exclusion des autres, lesquels se trouvent rejetés dans le prédicat pèle-mèle avec le verbe et tous les circonstants. (...) L'opposition du sujet et du prédicat masque en particulier le caractère interchangeable des actants, qui est à la base du mécanisme des voix active et passive".

Ceci dit, revenons-en aux actants et circonstants. Il est vrai qu'il est parfois difficile de délimiter clairement l'appartenance d'un noeud à l'une ou à l'autre catégorie. Toutefois, Tesnière propose deux critères sur lesquels s'appuyer : la forme et le sens.

Du point de vue de la forme, l'actant est le plus souvent un substantif (ou un substitut de celui-ci) alors que le circonstant est un adverbe ou un groupe prépositionnel équivalent à un adverbe.

Du point de vue du sens, l'actant fait corps avec le verbe, il est souvent nécessaire à la compréhension du procès que celui-ci exprime.

9.3.8. Nucléus

La dernière notion que nous exposerons montre bien à quel point Tesnière a réussi à construire une syntaxe qui tienne compte de l'interprétation sémantique sans trahir ses buts ni recourir à une hypothétique "intuition".

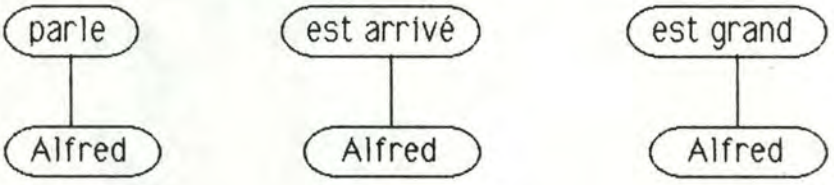
Il est des phrases dans lesquelles on est bien forcé de reconnaître que le noeud structural ne s'identifie pas avec le centre auquel aboutit la fonction sémantique. Tesnière s'est ainsi vu forcé d'élargir la notion de noeud et a créé un nouveau concept : le **nucléus**.

Le nucléus est un "carrefour", le siège de plusieurs fonctions, dont nous ne retiendrons que deux (il serait trop long de les exposer toutes) : la fonction structurale (le nucléus contient le noeud) et la fonction sémantique (le nucléus doit contenir un centre sémantique).

Le nucléus est représenté dans le stemma sous forme d'une ligne circulaire entourant le(s) mot(s) concerné(s).

Le ou les mots, car un nucléus peut être "dissocié", à savoir qu'il arrive que, dans une phrase, la fonction nodale porte sur un mot et la fonction sémantique sur un autre. C'est le cas, par exemple, pour les temps composés, ou l'association verbe "être" et attribut.

Exemple :

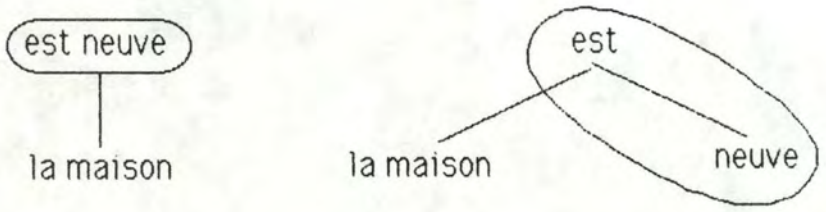


Si, dans la première phrase, fonction nodale et sémantique coïncident et sont représentées par le même mot, dans les deux exemples suivants, c'est "est" qui suporte la fonction nodale (comme auxiliaire de temps et verbe "substantif", respectivement) et "arrivé" ou "grand" qui, eux, supportent la fonction sémantique.

On peut déduire de ceci, que si le mot est l'unité linéaire de la phrase, c'est le nucléus qui en est l'unité structurale (c'est à dire cette fois, syntaxique et sémantique en même temps).

En pratique, bien sûr, et pour ne pas alourdir le stemma, on n'y fait figurer le cercle du nucléus que lorsque celui-ci est dissocié.

Autre remarque : il est possible de ne pas faire figurer les parties d'un nucléus dissocié au même niveau de l'arborescence (ceci pour faciliter le passage futur dans d'autres langues).



Les deux stemmas ci-dessus sont équivalents.

9.3.9. Remarque

Nous n'exposerons pas plus avant la grammaire de dépendance.

Précisons, cependant, que, à sa manière, Tesnière n'a pas négligé les phénomènes que Chomsky appelle des "transformations". Il y répond par les concepts de diathèse et de translation, notions plus affinées, qui permettent de rendre compte plus précisément des nuances que comportent le passif, le causatif, le changement de catégorie grammaticale, etc...

Expliquer tout ceci, cependant, nécessiterait tout un livre, et d'ailleurs nous n'en avons pas encore fait un usage direct dans notre travail.

9.4. Pourquoi avoir choisi la grammaire de dépendance au détriment du modèle transformationnel ?

9.4.1. Raisons purement pratiques

Dans un premier temps, parce que le choix de l'Esperanto comme représentation intermédiaire nous dispensait de fastidieuses transformations pour atteindre une structure profonde transférable d'une langue à l'autre.

Ensuite, parce que le modèle de dépendance nous offrait une manière simple et efficace de matérialiser à la fois la relation qui existe entre les mots d'une phrase et (chose absente chez Chomsky) la nature de cette relation, c'est à dire la fonction que remplit chaque syntagme dans la phrase; il nous suffit pour cela d'accoler des étiquettes à chaque ligne de connexion dans le stemma.

9.4.2. Au niveau théorique

Nous avons déjà mentionné certaines raisons de notre choix dans les chapitres traitant des deux grammaires.

Nous pourrions les résumer comme suit.

a) Chomsky ne s'intéresse réellement qu'à l'anglais, d'ailleurs ses théories n'ont pas bonne presse auprès des traducteurs [NIDA] alors que Tesnière est à la fois un comparatiste et un pédagogue qui fut quotidiennement confronté aux problèmes que pose la traduction, problèmes auxquels sa théorie tente de remédier.

b) Toutes les tentatives de Chomsky pour expliquer des phénomènes sémantiques par le biais du modèle transformationnel ont échoué. Or tout le monde est aujourd'hui d'avis qu'une traduction automatique de qualité ne sera possible que le jour où nous parviendrons à intégrer dans nos programmes un maximum de traitement sémantique.

De nombreux sémanticiens estiment qu'une théorie sémantique du langage doit s'appuyer sur un modèle syntaxique performant, et beaucoup s'accordent pour trouver en Tesnière non seulement le fondateur d'une telle démarche d'analyse syntaxique mais aussi le précurseur d'un modèle sémantique.

Pour citer J.P. Paillet, sémanticien québecois, "Tesnière présente d'intéressantes intuitions sur l'organisation sémantique", ou encore, "Si nous pouvons généraliser la notion de dépendance pour rendre compte de tous les types de connexions, nous aurons une notation plus souple que celle, linéaire, de la logique, qui apparaîtraît comme l'ordre linéaire du langage logique, notre notation correspondant à l'ordre structural".

Cela laisse un grand chemin à parcourir, mais nous croyons, nous aussi, que Tesnière nous a ouvert la bonne voie.

Chapitre 10 : Le Modèle des ATN

10.1. Introduction

Les "Augmented Transition Networks" ou ATN ont été développés par William Woods au début des années 70 [WOODS1], et sont utilisés depuis environ 15 ans, comme moyen de représentation d'une grammaire, dans différents systèmes de compréhension du langage naturel, ainsi que dans des systèmes questions-réponses, à la fois pour le texte et la parole.

Ils se sont avérés souples à utiliser, faciles à écrire et à corriger, capables de manipuler une grande variété de constructions syntaxiques, et faciles à intégrer à d'autres composants d'un grand système.

Il n'est pas nécessaire de savoir programmer un ordinateur pour écrire ou utiliser une grammaire écrite sous forme d'ATN, ce qui la rend facilement accessible aux linguistes.

Les ATN font partie, comme leur nom l'indique, de la famille des réseaux de transition ("Transition Networks").

10.2. Automates à états finis

Une grammaire est la définition formelle de toutes les phrases du langage, et se représente par un ensemble de règles d'écriture. Cependant, une grammaire peut aussi être vue comme la description d'une machine abstraite qui est capable d'accepter ou de générer uniquement les phrases qui appartiennent au langage. Ces machines abstraites sont appelées automates, et leur complexité est directement déterminée par la complexité de la grammaire à laquelle ils correspondent.

Ces automates se définissent en termes d'un ensemble d'états, d'un appareil en entrée qui peut accéder à un symbole à la fois, et d'une unité de contrôle qui peut examiner et lire ce symbole, et forcer l'automate à passer d'un état à un autre.

Les plus simples de ces automates sont appelés automates à états finis;

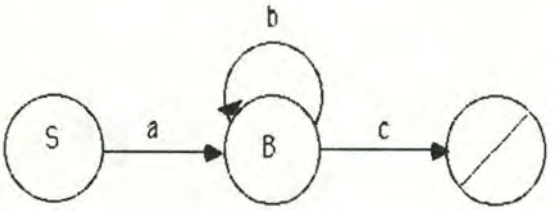
leur comportement est entièrement déterminé par le symbole courant qui est lu, et par l'état courant de la machine.

Soit par exemple la grammaire simple :

S -> aB
B -> bB
B -> c

qui génère le langage ab^*c.

L'automate à états finis qui lui correspond peut être représenté par le diagramme :



dans lequel les noeuds correspondent aux états, et les arcs aux transitions.

Les étiquettes sur les arcs représentent les conditions sur le symbole en entrée qui permettent de suivre une transition. Les noeuds contenant une barre diagonale sont appelés "états finaux".

Le passage des règles de réécriture aux automates est facile : les symboles non-terminaux (dans l'exemple, B) deviennent les états, et les symboles terminaux, les transitions.

Le comportement d'un automate peut être décrit de la manière suivante : l'analyse commence dans un état particulier appelé "état initial" (dans l'exemple, S); on examine le premier symbole de l'expression en entrée pour voir s'il existe une transition qui lui correspond; s'il en existe une, la machine suit cette transition, et passe à l'état suivant. On dit qu'elle "consomme" le symbole, puis passe au symbole suivant. Ce processus se déroule jusqu'à ce que l'automate ne trouve plus de transitions possibles. Si la machine se bloque alors qu'elle a atteint un état final, et que tous les symboles de l'expression en entrée ont été consommés, on dit que cette expression est acceptée (ou reconnue), et qu'elle appartient donc bien à la grammaire.

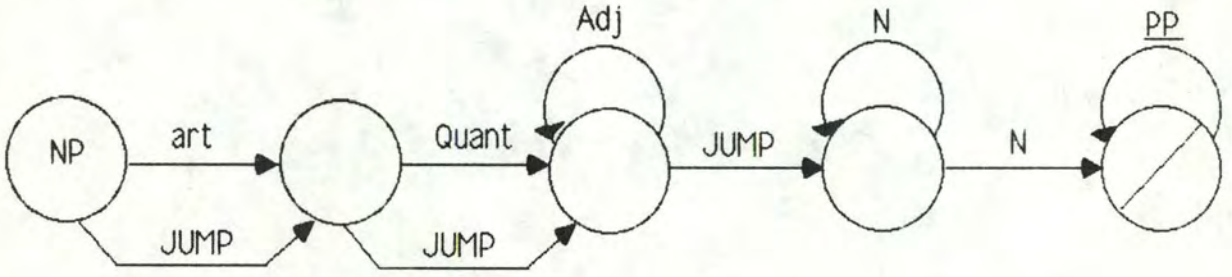
10.3. Transition Networks

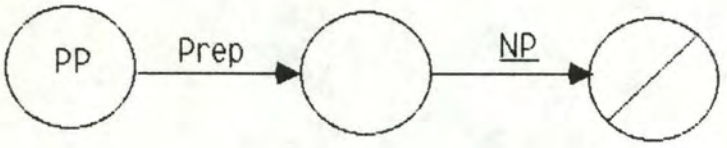
Soit les règles de réécriture suivantes qui sont une manière de représenter le groupe nominal (NP), et le groupe prépositionnel (PP) :

NP -> (Art) (Quant) Adj* N* N PP*
PP -> Prep NP
Art -> the |a |an...
Quant -> all |some...
Adj -> big | fat |nice...
N -> dog | girl |love...

où ce qui est entre parenthéses est optionnel, et ce qui est suivi d'un astérisque peut être présent plusieurs fois.

A ces règles correspondent les diagrammes de transition suivants :





où l'arc JUMP est un type d'arc qui peut être suivi sans aucune condition.

Cependant, le symbole PP est non-terminal, alors qu'il correspond à une transition dans l'automate de NP; comment peut-il donc être reconnu par cet automate ?

Pour résoudre ce problème, il suffit de considérer les deux automates comme un système ou un réseau.

Lorsque l'automate NP rencontre la transition PP, il passe temporairement le contrôle à l'automate PP, reportant sa décision jusqu'à ce que ce dernier lui renvoie l'information disant si oui ou non, un groupe prépositionnel (PP) peut être reconnu à la position courante actuelle dans l'expression en entrée.

Il est évident qu'un simple automate fini n'est plus suffisant pour traiter ce genre de système. Il faut lui ajouter un mécanisme récursif sous la forme d'une mémoire interne lui permettant de se souvenir d'où il vient (par quel automate il a été appelé) et comment retourner à cet automate quand il aura terminé son traitement.

Quand le contrôle est passé à un diagramme imbriqué, l'état courant de la machine est sauvé sur la pile, de telle sorte que l'analyse peut être poursuivie depuis ce même état, lorsque le contrôle est rendu au diagramme initial.

Ce type de système est appelé "**réseau de transition**" ou "transition network".

Une tentative d'accéder à la pile alors qu'elle est vide, et qu'on a déja traité le dernier caractère de l'expression en entrée, signifie que la phrase est reconnue par le réseau.

Il faut cependant remarquer que ces réseaux permettent uniquement d'accepter ou de refuser des phrases en entrée, selon une grammaire, mais ne fournissent aucun moyen pour construire la structure syntaxique de cette phrase. Or, le but d'un analyseur syntaxique est justement de fournir la structure syntaxique d'une phrase. Il est donc nécessaire de disposer d'un outil plus puissant que ces réseaux de transition de base, un outil offrant la possibilité de déplacer des fragments de la structure d'une phrase (de leur position courante dans la structure profonde de la phrase), de copier et de supprimer des fragments de structure, et de rendre les actions sur les constituants d'une phrase dépendants du contexte dans lequel ces constituants apparaissent.

C'est pourquoi ils ont été modifiés et on a obtenu ce qu'on appelle les "Augmented Transition Networks" ou ATN.

10.4. ATN

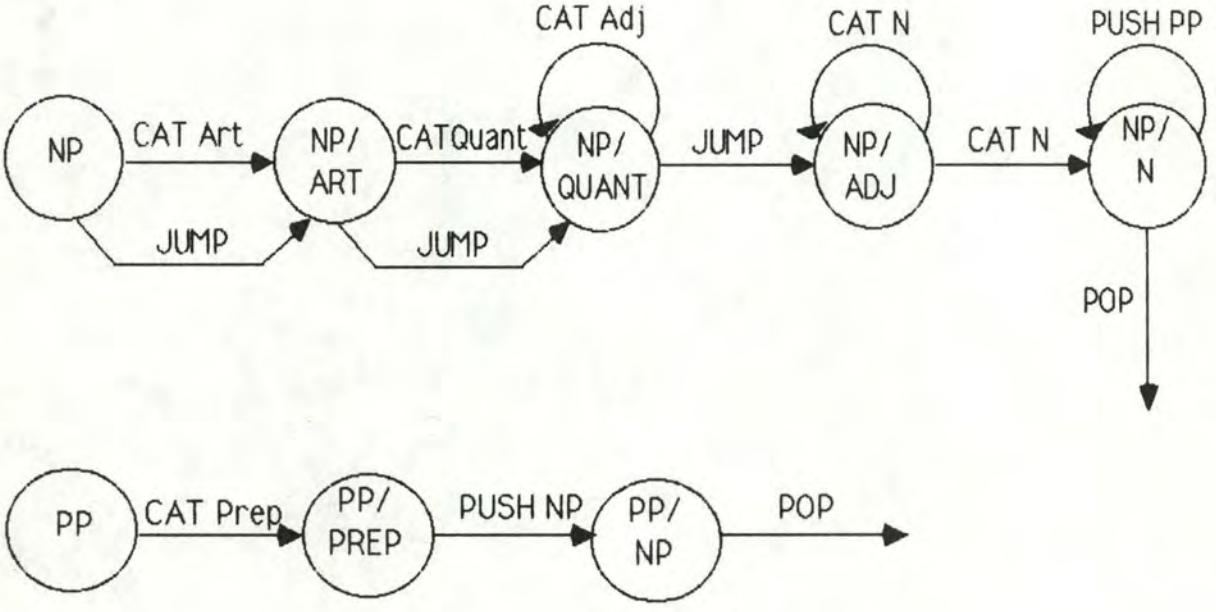
Dans le formalisme des ATN, le passage de contrôle à un moment donné se fait par une transition de type "PUSH" (par exemple, PUSH PP passe le contrôle au diagramme d'état représentant PP).

Pour les états finaux, on a créé un type de transition particulier appelé "POP", reflétant le fait que, quand un processus se termine, l'état original de la machine est rétabli à partir de la pile.

Un autre type de transition important est l'arc "CAT", qui signifie que le mot courant dans l'expression en entrée doit être de la catégorie syntaxique indiquée (et est "consommé" si l'arc est suivi).

Tout état est représenté par un cercle contenant le nom de l'état dans son milieu.

Avec les éléments dont on dispose jusqu'à présent, la grammaire proposée en 10.3. se représente de la manière suivante :



Pour les raisons exposées en conclusion de 10.3., il est nécessaire d'ajouter aux réseaux de transition classiques, une mémoire dans laquelle les détails des transitions précédentes peuvent être stockés et retrouvés, suivant des conditions sur des transitions ultérieures (afin de pouvoir reporter des décisions, en cours d'analyse, jusqu'à ce que ce qui vient plus loin dans la phrase soit connu).

Dans les ATN, cela a été réalisé en ajoutant à chaque niveau du réseau, un ensemble de registres, auxquels sont associées des fonctions prédéfinies pour les manipuler (les plus courantes sont la fonction d'assignement SETR, et la fonction GETR qui renvoit la valeur d'un registre).

On a également ajouté, à chaque arc, une condition arbitraire qui doit être satisfaite pour que l'arc soit suivi, ainsi qu'un ensemble d'actions qui permettent de construire des morceaux de structures, et qui sont exécutées si l'arc est suivi.

Tous ces mécanismes permettent à un ATN de construire une description partielle de la structure de la phrase au fur et à mesure qu'il analyse cette phrase, état par état.

Les parties de cette description sont mémorisées dans des registres, en cours d'analyse. Les actions de construction de structure sur les arcs, spécifient des changements dans le contenu de ces registres, en termes de leur ancien contenu, du contenu d'autres registres, du symbole courant en entrée, ou du résultat de calculs de niveaux inférieurs.

Les registres permettent donc de conserver des pièces de sous-structures qui seront éventuellement intégrées dans une structure plus grande; ceci est réalisé grâce à un fonction bien utile, "BUILDQ", qui permet de construire des morceaux de structure de phrase, sous forme de liste arborescente contenant des littéraux et des valeurs de registres [cfr. 10.5. d)].

A chaque état final est associé une ou plusieurs conditions qui doivent être satisfaites pour que l'état puisse rendre le contrôle à un niveau supérieur (causer un "pop"). Et à chacune de ces conditions, est associée une fonction qui calcule la valeur à retourner à l'état supérieur (par l'arc POP). En outre, un registre spécial, appelé *, (et qui contient habituellement le mot courant en entrée) contient le résultat du calcul du niveau inférieur (et qui a été appelé par un arc PUSH), c'est à dire ce qui est renvoyé par un arc POP.

En définitive, les actions sur les registres (utilisant SETR et BUILDQ) sont exécutées au fur et à mesure que l'analyse a lieu, de telle sorte que les contenus des registres reflètent constammant l'état le plus proche de ce que sera le résultat final de l'analyse.

Par exemple, quand on analyse la phrase "John was believed", lorsque le réseau rencontre le mot "was", il peut le considérer comme un verbe, le mettre dans le registre verbe, et extraire son temps (qu'il place dans le registre temps). Puis, quand il trouve le participe passé "believed", il peut forcer un "flag" du passif pour indiquer que la phrase est au passif, et placer le nouveau verbe "believe" dans le registre verbe, sans devoir revenir en arrière ("backtracking").

Il s'agit d'une des propriétés des ATN qui les rendent si interessants pour la recherche linguistique, et le développement de grammaires.

10.5. Représentation des ATN [BATES]

10.5.1. Description

<ATN> -> (<arc-set><arc-set>*)

<arc-set> -> (<nom-état><arc>*)

<arc> -> (CAT <nom-catégorie><test><action>* (TO <état-suivant>)) |
(PUSH <état><test><action>* (TO <état-suivant>)) |
(TST <test-arbitraire><test><action>* (TO <état-suivant>)) |
(JUMP <état-suivant><test><action>*) |
(POP <forme><test>) |
(WRD <mot><test><action>* (TO <état-suivant>)) |
(MEM <liste><test><action>* (TO <état-suivant>))

<action> -> (SETR <registre><forme>) |
(SENDR <registre><forme>) |
(LIFTR <registre><forme>) |
(SETRQ <registre><valeur>) |

(ADDL <registre><forme>)|
(ADDR <registre><forme>)|
(SENDRQ <registre><valeur>)|
(VERIFY <forme>)

<forme> -> *|
(GETR <registre>)|
(GETF <caractéristique><mot>)|
(APPEND <forme><forme>)|
(QUOTE <valeur>)|
(COND <predic1><e11><e12>.....<e1n>)
<predic2><e21>... ...<e2m>)
.....
<predic i><ei1>... ...<eij>)
(BUILDQ ....)

<predic> -> (NULLR <registre>)|
(CHECKF <caractéristique><valeur>)|
(CATCHECK <mot><nom-catégorie>)|
(OR <predic 1>...<predic n>)|
(AND <predic 1>...<predic n>)|
(EQUAL <forme1><forme2>)|
(APPARTIENT <regisre><liste>)

N.B: L'opérateur * signifie 0 ou plusieurs occurences de ce qui précède.

10.5.2. Les arcs

Le premier élément de chaque arc indique son type, le troisième est un test arbitraire (une condition) qui doit être satisfait afin que l'arc soit suivi, et l'interprétation du deuxième dépend du type d'arc.

-CAT,PUSH,JUMP,POP [cfr. 10.4.]

-TST : permet un test arbitraire pour déterminer si un arc doit être suivi (en plus de la condition sur l'arc).

-WRD : spécifie le mot exact qui est requis en entrée.

-MEM : le mot requis doit appartenir à la liste donnée pour que l'arc soit suivi.

réseau de ce niveau d'être considéré comme une sous-routine auquel on passe des paramétres à l'aide de SENDR).

10.5.4. Les formes

-LEX: contient le mot courant en entrée tel quel.

-* : réfère l'élément courant en entrée; sur un arc CAT, c'est la racine du mot en entrée; sur un arc PUSH, c'est le mot courant pour le test et les pré-actions. Pour les autres actions, c'est la valeur retournée par le niveau inférieur (arc POP) appelé par le PUSH.

-GETR : retourne le contenu courant du registre indiqué.

-GETF : vérifie l'entrée du dictionnaire du mot indiqué, et retourne la valeur de la caractéristique spécifiée (trouvée dans le dictionnaire).

-BUILDQ : c'est l'action qui permet de construire des morceaux de structures. Elle prend un fragment de structure contenant des constantes et des marques spéciales (* ou +), ainsi qu'une série de noms de registres, et renvoit la structure obtenue en substituant les contenus des registres aux marques spéciales (le premier registre remplaçant la premier + rencontré, le deuxième registre remplaçant le deuxième +, etc.).
Exemple :
(BUILDQ (S + + +) SUBJ AUX VP) renverrait (S (NP John) does (VP (V like) (NP Mary))), si le registre SUBJ contient"(NP John)", AUX contient "does", et VP contient "(VP (V like) (NP Mary))".

-APPEND : ajoute la deuxième forme à la première.

-QUOTE : donne la valeur indiquée (sans être évaluée).

-COND : permet d'écrire une instruction conditionnelle (les <e> peuvent être des actions ou des formes).

10.5.5. Les prédicats

-NULLR : vrai si le registre nommé n'a jamais eu de valeur, ou s'il a été mis à NIL.

-OR : vrai si un des prédicats $pred_i$ est vrai.

-AND : vrai si tous les prédicats $pred_i$ sont vrais.

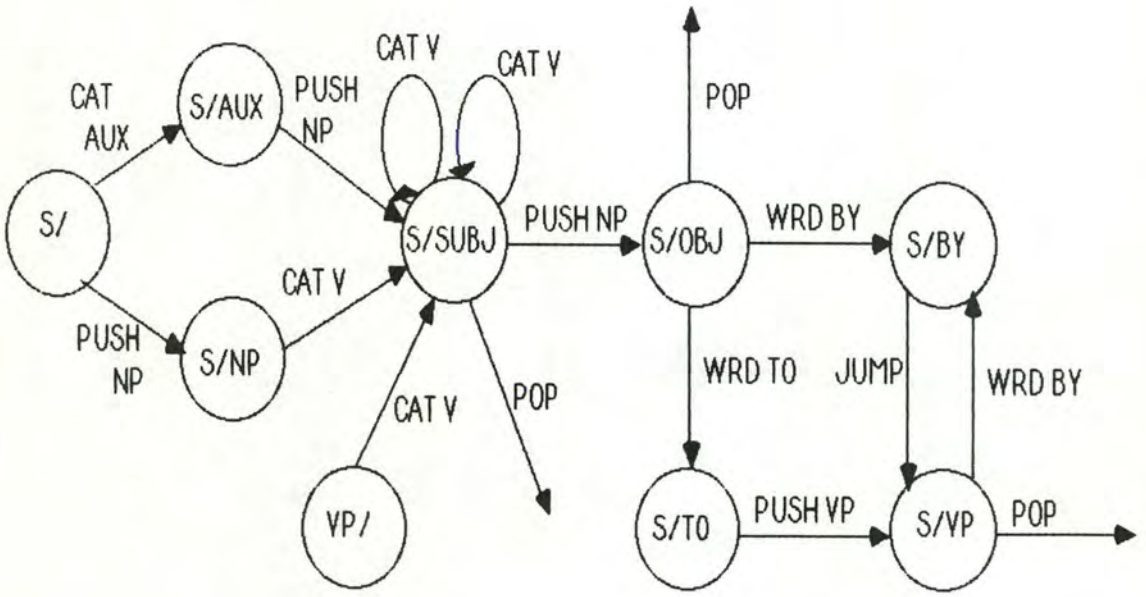
-EQUAL : vrai si forme1 et forme2 ont la même valeur.

-APPARTIENT : vrai si la valeur de registre appartient à liste.

-CHECKF : vrai si la valeur de la caractéristique pour le mot courant correspond à la valeur indiquée.

-CATCHECK: vrai si le mot indiqué est de la catégorie syntaxique donnée.

10.5.6. Exemple



Voici la représentation des arcs et des états de la grammaire présentée ci-dessus :

```
(S/
  (CAT AUX T  /* T est un prédicat qui est toujours vrai */
        (SETR V *)
        (SETR TNS (GETF TENSE))
        (SETR TYPE (QUOTE Q))
        (TO S/AUX))
  (PUSH NP/ T
        (SETR SUBJ *)
        (SETR TYPE (QUOTE DCL))
        (TO S/NP))


(S/AUX
  (PUSH NP/ T
        (SETR SUBJ *)
        (TO S/SUBJ))

(S/NP
  (CAT V T
        (SETR V *)
        (SETR TNS (GETF TENSE))
        (TO S/SUBJ))

 (S/SUBJ
   (CAT V (AND (GETF PPRT) (EQ (GETR V) (QUOTE BE)))
        (HOLD (GETR SUBJ))
        (SETR SUBJ (BUILDQ (NP (PRO SOMEONE))))
        (SETR AGFLAG T)
        (SETR V *)
        (TO S/SUBJ))

   (CAT V (AND (GETF PPRT) (EQ (GETR V) (QUOTE HAVE)))
        (SETR TNS (APPEND (GETR TNS)(QUOTE PERFECT)))
        (SETR V *)
        (TO S/SUBJ))

   (PUSH NP/ (TRANS (GETR V))
        (SETR OBJ *)
        (TO S/OBJ)))
```

```
  (POP ((BUILDQ (S++(TNS+)(VP(V+)+)) TYPE SUBJ TNS V) INTRANS (GETR V))

(S/OBJ
  (WRD BY (GETR AGFLAG))
       (SETR AGFLAG NIL)
       (TO S/BY))

  (WRD TO (S-TRANS (GETR V))
       (SENDR SUBJ (GETR OBJ))
       (SENDR TNS (GETR TNS))
       (SENDR TYPE (QUOTE DCL))
       (TO S/TO))

  (POP ((BUILDQ (S + + (TNS +) (VP (V +) +) ) TYPE SUBJ TENSE V OBJ)  T)

(S/TO
  (PUSH VP/   T
       (SETR OBJ *)
       (TO S/VP))

(S/BY
  (JUMP S/VP T))

(S/VP
  (WRD BY (GETR AGFLAG)
        (SETR AGFLAG NIL)
        (TO S/BY))

  (POP  ((BUILDQ (S ++ (TNS+) (VP (V+)+)) TYPE SUBJ TENSE V)  T)

(VP/
  (CAT V (GETF UNTENSED)
        (SETR V *)
        (TO S/OBJ)))
```

Prenons par exemple la phrase "John likes Mary".

On part de l'état initial de la grammaire, S/; deux arcs quittent cet état : CAT AUX et PUSH NP. Le premier ne peut pas être suivi, car le mot courant (le premier mot de la phrase, soit "John") n'est pas de la catégorie syntaxique

auxiliaire. Le second fait appel au diagramme NP/ pour vérifier si on a un groupe nominal, à partir du mot courant; le diagramme NP/ renverra (par un arc POP) la structure (NP (N John)), qui signale qu'il a trouvé un groupe nominal constitué du nom "John"; cette structure est placée dans le registre SUBJ, et le registre TYPE va contenir "DCL" pour indiquer que la phrase est affirmative.

Le contrôle est alors passé à l'état se trouvant à l'extrémité de l'arc PUSH NP, dans le diagramme S/, soit S/NP. De cet état part un seul arc, CAT V; il faut donc que le mot courant soit de la catégorie verbe. Comme il s'agit de "likes", l'arc peut être suivi, et le verbe "like" est placé dans le registre V; de plus, le registre TNS va contenir le temps du verbe, c'est à dire "présent".

Si le mot courant n'avait pas été un verbe, l'arc n'aurait pas pu être suivi, et comme aucun autre arc n'aurait pu être emprunté, on en aurait conclu que la phrase ne respectait pas la grammaire.

Nous sommes maintenant à l'état S/SUBJ, duquel partent divers arcs CAT V (mais le mot courant n'est plus un verbe, il s'agit de "Mary", un nom); l'arc POP ne peut lui non-plus être suivi car il reste un mot à consommer. Le dernier arc possible est PUSH NP; tout comme pour le mot "John", l'appel au diagramme NP/ va renvoyer la structure (NP (N Mary)), qui va être placée dans le registre OBJ.

Nous arrivons à l'état S/OBJ; un arc POP quitte cet état, et comme il ne reste plus de mot à consommer dans la phrase, nous empruntons cet arc qui renvoie la structure syntaxique complète de la phrase :

(S DCL (NP (N John)) (TNS present) (VP (V like) (NP (N Mary))))

Chapitre 11: Formalisation d'une grammaire Esperanto

11.1. Ecriture d'une grammaire

Nous nous sommes inspirés de la grammaire écrite dans le cadre du projet DLT par la firme néerlandaise BSO [SCHUBERT] [WITKH], qui utilisait également la grammaire de dépendance de Tesnière.

11.1.1. Les catégories grammaticales

La première étape de l'écriture de la grammaire Esperanto consiste à faire l'inventaire des catégories grammaticales des mots que compte l'Esperanto. Nous rappellerons tout d'abord que l'Esperanto comporte deux sortes de mots : les mots porteurs de sens et les outils grammaticaux.

Nous avons recensé quatre catégories de mots porteurs de sens : verbe, substantif, adjectif, et adverbe, qui sont formés par agglutination d'affixes autour de racines sémantiques [cfr 3.2.2].

Les outils grammaticaux sont énumérables par catégorie, et figurent tous dans le dictionnaire tels quels. Ils appartiennent à huit catégories différentes : les adjectifs corrélatifs (kia, tia, alia,...), les adverbes (tial, ne,...), les pronoms, qui peuvent être soit personnels (mi, li,...), soit possessifs (mia, lia, tia,...), soit relatifs (kiu,kio), l'article (la), les numéraux (unu, du,...), les prépositions (al, dum,...), les conjonctions de coordination (nu, sed,...), et enfin les conjonctions de subordination (ke, kiel, se,...).

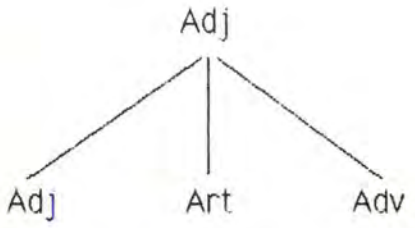
Nous avons donc finalement dix catégories grammaticales différentes.

11.1.2. Recherche des dépendants

La deuxième étape consiste à considérer chacune des catégories grammaticales comme le noeud principal du stemma, et à rechercher l'ensemble de tous ses dépendants possibles.

Cela revient à établir les rapports de hiérarchie de toute catégorie, c'est à dire les catégories grammaticales qu'une catégorie donnée peut gouverner.

Par exemple, un adjectif peut gouverner (entre autres) un autre adjectif, un article, ou un adverbe, ce qu'on représente par :



11.1.3. Analyse des connexions

A partir des dix arbres de dépendance construits au paragraphe précédent, il faut déterminer quel rôle joue chaque dépendant par rapport à son régissant. Cela revient à analyser, pour tout arbre, les connexions qui unissent le régissant à ses dépendants. Ces connexions sont en fait des fonctions syntaxiques.

Par exemple, dans le cas de l'adjectif, un autre adjectif ou un article joue le rôle de déterminant par rapport à l'adjectif-régissant, alors qu'un adverbe joue le rôle de circonstant adverbial.

Nous présentons ci-dessous les différentes fonctions que nous avons défini (chaque fonction peut prendre la forme d'un des syntagmes appartenant à sa représentation linéaire) :

-**Valence 1** : c'est le sujet de la phrase, ou encore le prime-actant défini par Tesnière.

Val1 = GNom | GPron | GAdj | GNum | PRel | PSub (ke) | PInf

-**Valence 2** : c'est le complément d'objet direct, ou second-actant.

Val2 = GNom | GPron | GAdj | PRel |GNum | PSub(ke) | PInf

-**Valence 3** : c'est le complément d'objet indirect, ou tiers-actant. Il est toujours introduit par Al, Pri, De ou Je.

Val3 = GNom | GPron | GAdj | PRel | GNum | PSub(ke) | PInf

-**Attribut du verbe** :

Attr = GNom | GPron | GAdj | GNum | GPrep(por) | PSub(ke)

-**Complément infinitif** :

CInf = PInf

-**Circonstant adverbial** :

CAdv = GAdv

-**Circonstant prépositionnel** :

CPrep = GPrep

-**Circonstant propositionnel** :

CProp = PSub

-**Déterminant 1** : déterminant (précise ou modifie le sens du mot) qui se place avant le mot qui le gouverne.

Det1 = article | GAdj | GNum | GPron

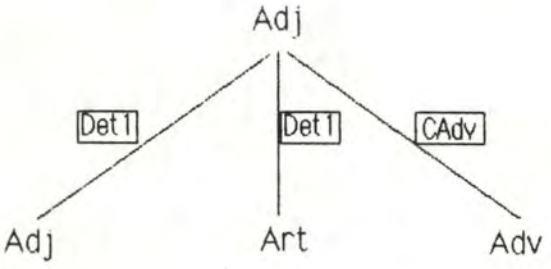
-**Déteminant 2** : déterminant qui se place après le mot qui le gouverne.

Det2 = GAdj | PRel

où les notations utilisées ont les significations suivantes : GNom : Groupe Nominal, GPron : Groupe Pronominal, GAdj : Groupe Adjectival, GNum : Groupe Numéral, GPrep : Groupe Prépositionnel, GAdv : Groupe Adverbial, PRel : Proposition Relative, PSub : Proposition Subordonnée, PInf : Proposition Infinitive.

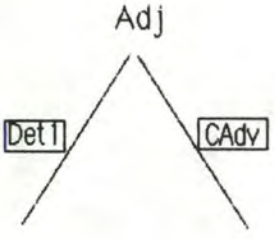
On obtient ainsi dix nouveaux arbres, qui exposent les divers types de dépendants qu'un mot d'une catégorie grammaticale donnée, peut gouverner.

L'arbre de l'adjectif devient ainsi :



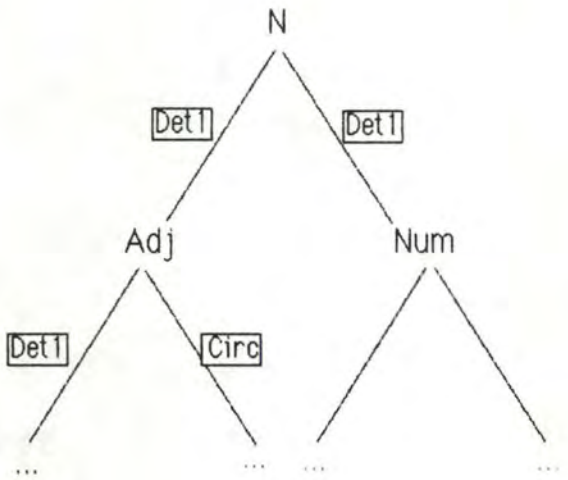
Etant donné que dans un même stemma, on peut avoir plusieurs arcs portant le même nom de fonction (ex : Det pour Adj), on va simplifier ces stemmas; on va obtenir dix nouveaux arbres ne contenant plus que les traits de connexion, c'est à dire une branche par fonction (et plus le nom des catégories à leur extrémité). [cfr Annexe 1].

On obtient donc, pour l'adjectif :



Comme nous l'avons expliqué en 9.3.5. , le verbe constitue le noeud central de la phrase. Cependant, pratiquement n'importe quel mot peut être régissant d'autres mots; on appelle **syntagme** le groupe de mots gouverné par un mot.

Par exemple, le syntagme nominal est l'ensemble des mots que peut régir un nom (on l'appelle aussi groupe nominal) :



Le syntagme nominal peut contenir lui-même plusieurs syntagmes (par ex, le syntagme adjectival), et s'obtient en appliquant récursivement les arbres qui correspondent à chacun des syntagmes.

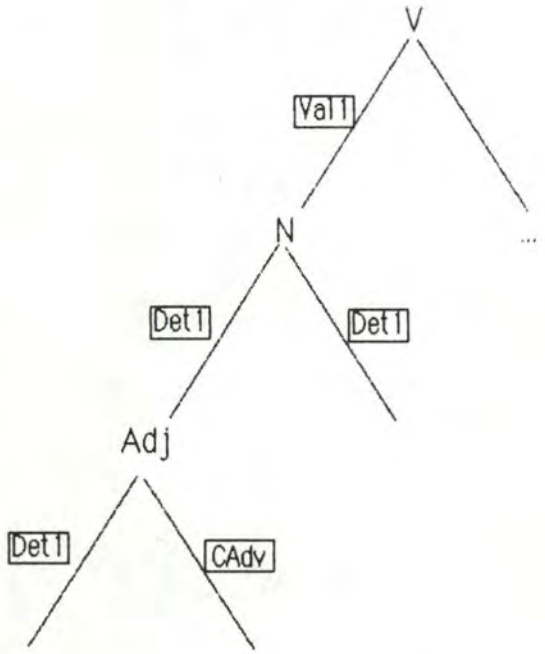
11.1.4. Passage de l'ordre structural à l'ordre linéaire

Nous disposons maintenant des dix arbres élémentaires pour chaque catégorie grammaticale. Or, l'ordinateur traite du texte et non pas des structures sous forme d'arbres. Il faut donc transposer ces structures en texte linéaire.

Passer à l'ordre linéaire revient à trouver la représentation linéaire de la phrase.

Pour réaliser cela, on utilise les arbres élémentaires. On part d'un stemma, et chaque fois qu'un noeud fait partie d'un des dix stemmas, on accole ce stemma à ce noeud.

Exemple :



Adj intervient dans le stemma du nom (comme Det1) et dans celui du verbe (comme Val1); on ajoute alors le stemma du nom au dessus de celui de l'adjectif, avec la branche Det1 qui aboutit à Adj. On fait la même chose avec le stemma du verbe (on retranscrit le stemma de l'adjectif à un niveau supérieur -même niveau que le nom- et on le rattache au verbe V. On itère ce processus tant que c'est possible. L'arbre obtenu grossit ainsi à chaque itération.

Lorsqu'on a l'arbre final, on le parcourt en partant du verbe, qui en est le noeud principal.

On obtient la représentation linéaire de la phrase en articulant tous les syntagmes qui supportent les fonctions, à partir du verbe, ce qui donne :

PRINC = Val1 | Verbe | Val2 | Val3 | Attr | CInf | CAdv | CPrep | CProp,

Ensuite, on réapplique le même procédé pour obtenir la forme linéaire de chacun des syntagmes, en partant du mot principal du syntagme (par ex., du nom pour un syntagme nominal).

On obtient :

-GNom = Det1 | NOM | Det2 | Circ(s)

-GPrep = PREP | GNom | CAdv
= PREP | GPron | CAdv
= PREP | GNum | CAdv
= PREP | PInf | CAdv
= PREP | CPrep | CAdv
= PREP | PSub(Ke,Chu) | CAdv

-GNum = Det1 | NUM | Det2 | CPrep | GNum

-GPro = Det1 | PRO | Det2 | CAdv | CPrep | CProp

-GAdv = ADV | CInf | CAdv | CPrep | CProp
= ADV | Val2 | Val3 | Circ(s)
= ADV | Val2 | Attr | Circ(s)
= ADV | Attr | Circ(s)
= ADV | CInf | Circ(s)

-GAdj = Det1 | CAdv | ADJ | Circ(s)
= ADJ | Val2 | Attr | Circ(s)
= ADJ | Attr | Circ(s)
= ADJ | Val2 | Val3 | Circ(s)
= ADJ | CInf | Circ(s)

-PInf = INF | Val2 | Val3 | CInf | Circ(s)
= INF | Val2 | Attr | Circ(s)
= INF | Attr | Circ(s)

-PSub = SUBJ | PRINC

-PRel = (Kiu, Kio, Kiel) | VERBE | Val2 | Val3 | Attr | Circ(s)
= (Kiun, Kion) | PRINC
= (Al, De, Pri, Je) | (Kiu, Kio) | PRINC

11.2. Formalisation sous forme d'ATN

11.2.1. Construction des ATN

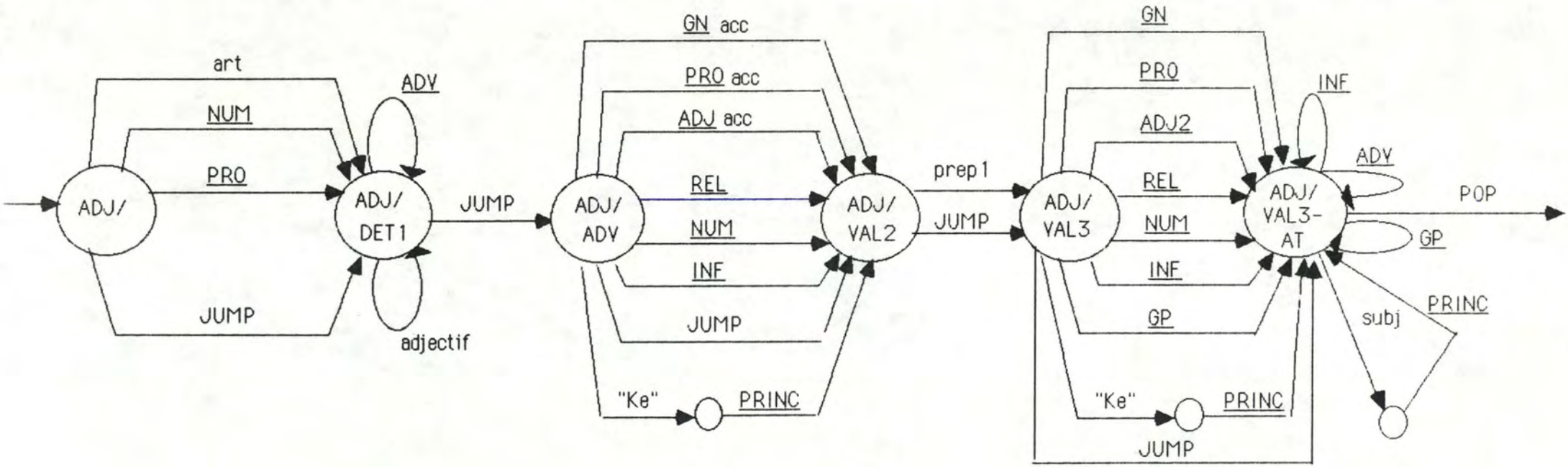
On part des règles du groupe adjectival, et on considère que ADJ/ est l'état initial du diagramme du groupe adjectif. Pour passer à l'état suivant, il faut une construction de type Det1 ; Det1 peut être un des trois syntagmes Art, GNum, et GPron repris dans la règle de Det1, et on crée donc les transitions correspondant à ces syntagmes ("CAT art", "PUSH NUM", et "PUSH PRO"), plus un arc JUMP car il est possible qu'il n'y ait pas de Déterminant 1 au début d'un groupe adjectival. On nomme l'état suivant ADJ/DET1.

Ensuite, on peut trouver un groupe adverbial (arc "PUSH ADV"), qui doit nécessairement être suivi d'un adjectif, d'où on place un arc "CAT adjectif" et pas d'arc JUMP, et on appelle l'état suivant ADJ/ADV.

On continue à progresser dans les règles, et on se rend compte qu'il peut y avoir les différentes valences et l'attribut (même constructions que pour la phrase principale PRINC), ainsi que les circonstants : on crée donc les transitions correspondantes... Le diagramme se termine évidemment par un arc POP.

On obtient ainsi le diagramme de transition du groupe adjectif (page suivante).

NB : Les autres diagrammes de transition sont repris en annexes [cfr. ANNEXE 2].



prep1 = (Al, De, Pri, Je)

11.2.2. Encodage des ATN

Les ATN obtenus sont sous forme graphique, et ne sont donc évidemment pas directement compréhensibles par un ordinateur. La dernière étape consiste donc à les convertir en une forme lisible par un ordinateur.

Nous avons choisi, pour réaliser cet encodage, de représenter chaque état sous forme d'une liste encadrée par des parenthèses (contenant son nom, et la description de ses arcs), à l'intérieur de laquelle tout arc est lui-même décrit entre deux parenthèses (son nom, ses conditions, ses actions, et le nom de l'état suivant).

Ces descriptions sont contenues dans un fichier, et constituent une donnée du programme d'analyse syntaxique [cfr. Chap.12].

Voici, à titre d'exemple, le début de la description du groupe adjectif :

```
(ADJ/
  (CAT article T
    (SETR DET1 (BUILDQ((ART *))))
    (TO ADJ/DET1))
  (PUSH NUM/ NUM-START
    (SETR DET1 *)
    (TO ADJ/DET1))
  (PUSH PRO/ PRO-START
    (SETR DET1 *)
    (TO ADJ/DET1))
  (JUMP ADJ/DET1 T
    (SETR DET1 NIL)))

(ADJ/DET1
  (PUSH ADV/ T
    (COND ((NULLR CADV)(SETR CADV *))
          (T (ADDR CADV *)))
    (TO ADJ/DET1 T))
  (CAT adjectif T
    (COND ((NULLR ADJ)(SETR ADJ LEX))
```

```
      (T (ADDL ADJ LEX)))
   (TO ADJ/DET 1))
  (JUMP ADJ/ADV T))

(ADJ/ADV
  (PUSH GN/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
     (PUSH PRO/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
     (PUSH REL/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
     (PUSH NUM/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
     (PUSH INF/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
   (WRD Ke (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 (BUILDQ((SUBJ *))))
    (TO ADJ/KE2))
  (JUMP ADJ/VAL2 T
    (SETR VAL2 NIL)))
```

N.B : le fichier complet contenant cette description des ATN se trouve en annexes [cfr. ANNEXE3].

Chapitre 12 : L'analyse syntaxique

12.1. Analyse syntaxique avec les ATN

12.1.1. Introduction

Il est important de bien distinguer une grammaire ATN de l'analyseur syntaxique de cette grammaire. Un analyseur syntaxique est une procédure qui permet de découvrir comment est construite une phrase, c'est à dire qui assigne une structure à cette phrase, après l'avoir identifié ("reconnue").

Il existe différents algorithmes d'analyse syntaxique qui peuvent être utilisés pour une grammaire ATN -en fait, presque tout algorithme d'analyse syntaxique pour une grammaire context-free peut être adapté pour les ATN, en y ajoutant un mécanisme pour représenter et manipuler les contenus des registres, et pour exécuter les tests et les actions sur les arcs.

Certains analyseurs sont des interpréteurs [EARLEY], d'autres sont des compilateurs [BURTON].

Parmi tous les analyseurs que nous avons étudié, celui qui nous apparaissait comme le plus puissant était un analyseur écrit par Woods [WOODS2]; il a en effet déja été utilisé avec succés dans plusieurs applications [LUNAR], [PSYCHO],... et il peut être considéré comme le meilleur pour les grammaires ATN.

12.1.2. Principe de fonctionnement de l'analyseur de Woods

L'analyseur de Woods est un interpréteur du type "top-down", basé sur une liste d'alternatives, chacune d'entre elles contenant toute l'information nécessaire pour redémarrer l'analyse du point où l'alternative avait été créée.

Etant donné qu'une grammaire ATN est essentiellement un type d'automate à mémoire "push-down", augmenté par des registres, des actions de manipulation de ces registres, et des conditions, et est donc une sorte de machine abstraite , la manière la plus naturelle de penser à son fonctionnement est en termes de **configuration machine instantannée**, et de **fonction de transition** (fonction qui calcule les successeurs d'une configuration donnée).

Et cela d'autant plus que la grammaire de réseau de transition est une machine non déterministe (la configuration successeur d'une configuration donnée n'est pas déterminée univoquement).

12.1.3. Configuration

Une configuration peut être vue comme une photo instantanée de l'état courant de l'analyse, et consiste en la liste:

(string weight state stack regs hold path)

où:

-string indique à quel endroit la configuration se trouve dans l'expression analysée.

-weight est la probabilité estimée que la configuration soit la bonne.

-state est l'état du réseau dans lequel se trouve la configuration.

-stack rappelle les calculs de niveau supérieur dans lesquels la configuration est imbriquée (contient les *regs* et le *state* du niveau supérieur, et les actions en suspens sur son arc PUSH).

-regs est l'ensemble courant des contenus des registres.

-hold est la liste hold courante.

-path est un enregistrement complet qui indique comment cette configuration a été atteinte (quelle était la configuration précédente, quel arc a été utilisé pour arriver à cette configuration, et quel était le contenu du registre * pour cette configuration).

12.1.4. La fonction de transition

C'est une fonction abstraite qui détermine les configurations successeurs possibles d'une configuration donnée, et qui est implémentée par une fonction appelée STEP.

Quand on arrive dans un état au cours de l'analyse d'une phrase, avec une grammaire ATN, il y a en général plusieurs arcs qui quittent cet état, et plusieurs de ces arcs peuvent être suivis.

La grammaire ATN est d'une grande souplesse du fait qu'elle permet d'ordonner les arcs d'un état donné, et d'organiser la structure du réseau, afin de décider quel arc est le plus probable et de réduire le nombre d'arcs qui quittent un état.

Cependant, il restera finalement toujours une certaine ambiguité, qui exige soit de suivre différentes alternatives en paralléle, soit de suivre une seule alternative qui peut s'avérer mauvaise, et d'être alors contraint de faire du "backtracking"...

En définitive, la fonction STEP devra assigner autant de successeurs à une configuration donnée, qu'il y a d'arcs différents qui peuvent être suivis.

12.1.5. Stratégie de l'analyseur syntaxique

L'analyse d'une phrase selon une grammaire de réseau de transition consiste essentiellement à trouver une alternative ("COMPUTATION") qui réussisse, dans l'espace des alternatives possibles caractérisées par la machine abstraite. Dans la plupart des algorithmes d'analyse syntaxique, c'est réalisé par une énumération exhaustive de tout l'espace des alternatives possibles.

L'analyseur écrit par Woods considère plutôt qu'il s'agit d'un travail de recherche; il essaie de trouver l'alternative de la machine abstraite qui correspond à l'analyse la plus probable, mais sans devoir énumérer exhaustivement l'entièreté de l'espace de recherche.

Il recherche l'espace des possibilités dans un ordre arbitraire, utilisant toutes sortes d'informations disponibles pour suggérer quelle alternative essayer; par exemple, essayer d'abord une alternative, la poursuivre pendant un moment, puis la suspendre et travailler avec une autre, comparer ces deux alternatives pour décider laquelle est la plus probable...

La stratégie de base utilisée est la suivante: l'analyseur essaie les arcs quittant un état dans l'ordre dans lequel ils sont répertoriés dans le programme, et quand il trouve un arc qu'il peut suivre, il génére une entrée dans une liste d'alternatives appelée ALTS, qui contient toutes les informations nécessaires pour que le système puisse reprendre plus tard son analyse, de l'endroit où l'alternative a été créée.

Ainsi, les entrées peuvent être suivies dans n'importe quel ordre (et pas seulement dans l'ordre Last-In-First-Out), ce qui permet au concepteur de la grammaire d'ordonner les arcs quittant un état dans l'ordre dans lequel il veut les essayer.

La situation générale est que l'analyseur syntaxique a un ensemble de configurations qu'il est en train de suivre activement en paralléle, et une liste d'alternatives en attente, qui ne seront essayées que quand il sera à court de configurations actives.

12.1.6. Importance des registres

Sur tout arc, peu de registres sont susceptibles d'être modifiés, mais le nombre total de registres utilisés dans la grammaire peut être grand, et ne devrait être soumis à aucune limite finie.

Au contraire, le concepteur de la grammaire devrait toujours pouvoir inventer un nouveau registre.

Si on devait allouer une adresse dans la configuration, à chacun des registres, la représentation d'une configuration prendrait un grand nombre de mots-machine qui seraient, pour la plupart, copiés inchangés d'un configuration à ses successeurs.

Pour éviter cela, on va plutôt partager au maximum le contenu des registres communs à différentes configurations, et on va, dans ce but, représenter les contenus des registres par une liste de paires nom de registre/valeur de registre.

La fonction GETR (qui donne le contenu du registre indiqué) va repérer dans la liste un nom de registre spécifié, et prendre la valeur qui lui est associée.

La fonction SETR (qui assigne une valeur au registre indiqué) ne va pas repérer dans la liste le nom du registre spécifié, mais ajoute simplement une nouvelle paire nom/valeur **au début** de la liste (pouvant ainsi cacher une ancienne paire du même nom). Les autres configurations qui pointent au-delà de la paire ajoutée, ne sont pas affectées par ce changement.

Chaque processus peut ajouter des informations à la liste, et ne mémoriser que son nouveau pointeur vers le début de la liste; ainsi, tout processus ne peut

voir que son seul chemin depuis la paire indiquée par son pointeur, jusqu'à la fin de la liste.

On peut reprocher à cette manière de traiter la liste des registres qu'elle alourdit et ralentit l'accès à un registre pour consultation; cependant, l'affectation d'une valeur à un registre est accélérée, et, on a remarqué que lors de l'analyse syntaxique, les registres sont souvent remplis, alors qu'ils ne sont seulement examinés que pour des vérifications occasionnelles dans les conditions sur les arcs, et comme données pour les fonctions qui construisent les structures sur les arcs POP.

12.2. Architecture et stratégie de notre analyseur

Nous présentons fig.12.1 l'architecture de notre analyseur.

La fonction PARSER (de niveau maximum) est appelée avec la phrase à analyser comme argument. Elle établit une configuration initiale, (à l'état initial de la grammaire, avec *stack* et *regs* vides). Elle appelle ensuite la fonction LEXIC qui réalise l'analyse lexicale de l'expression pour déterminer le mot Esperanto suivant, et sa racine sémantique.

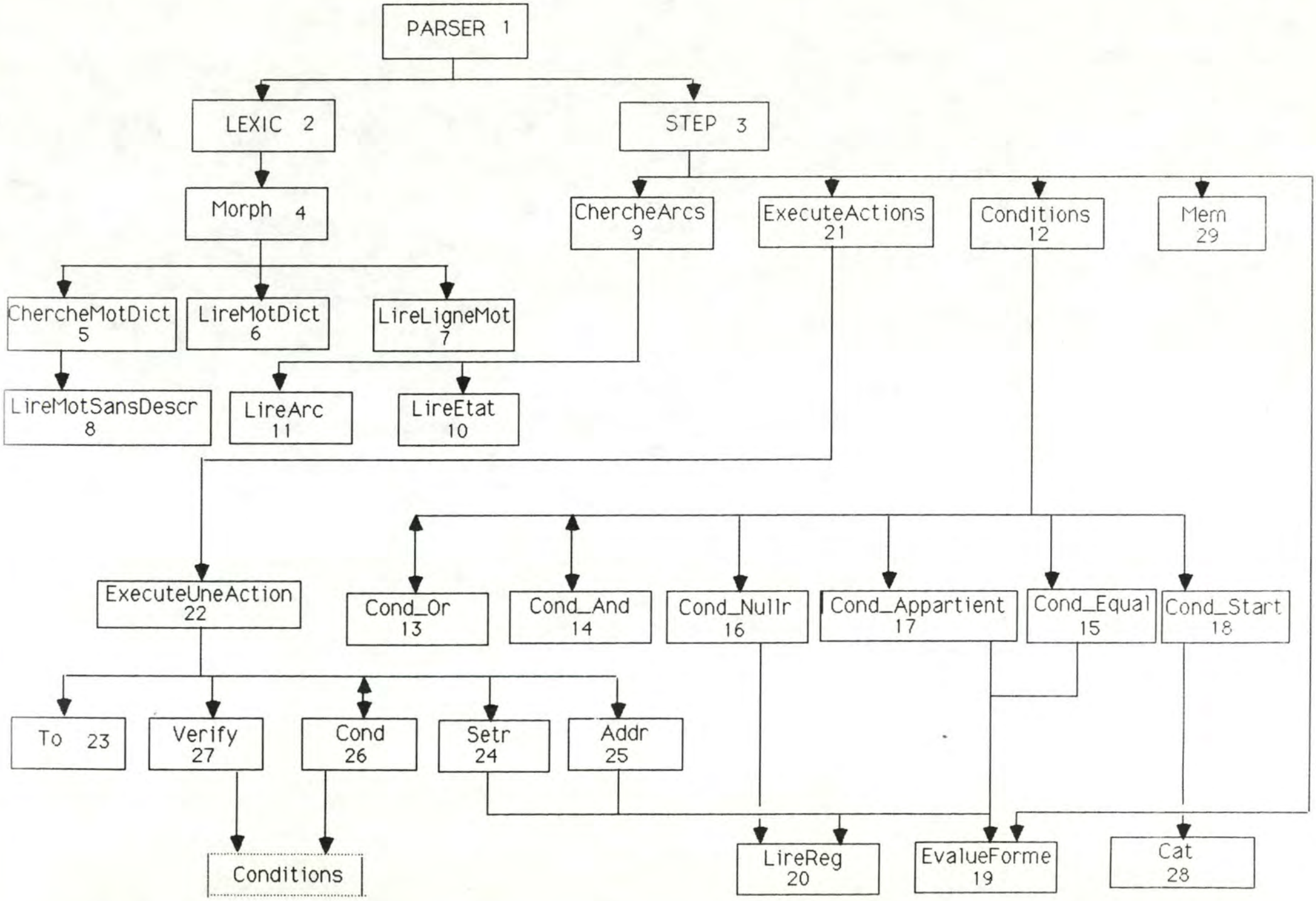
STEP, comme on l'a deja dit, est la fonction de transition qui calcule les configurations successeurs des configurations actives; elle prend celles-ci dans la liste des configurations actives, et y place ensuite les successeurs.

Elle utilise la routine CAT pour accéder aux entrées du dictionnaire et décider si le mot analysé est membre d'une catégorie syntaxique donnée.

PARSER boucle sur STEP et LEXIC tant que de nouvelles configurations sont créées. Lorsque la liste des configurations actives devient vide, PARSER doit activer une des alternatives créées par STEP, et appartenant à la liste des alternatives; étant donné que les entrées de cette liste contiennent toute l'information nécessaire pour redémarrer l'analyse à partir de n'importe quelle alternative, PARSER peut utiliser n'importe quelle stratégie pour sélectionner l'alternative suivante.

Nous avons décidé, pour des raisons d'efficacité, de choisir la première alternative de la liste des alternatives, qui correspond à la première des alternatives créées pour la dernière configuration traitée (et qui a échoué).

Quand STEP atteint la fin de la phrase avec une pile vide, et rencontre un arc POP, la structure obtenue est placée dans une liste (PARSES).


PARSER 1
LEXIC 2
STEP 3
Morph 4
ChercheArcs 9
ExecuteActions 21
Conditions 12
Mem 29
ChercheMotDict 5
LireMotDict 6
LireLigneMot 7
LireMotSansDescr 8
LireArc 11
LireEtat 10
ExecuteUneAction 22
Cond_Or 13
Cond_And 14
Cond_Nullr 16
Cond_Appartient 17
Cond_Equal 15
Cond_Start 18
To 23
Verify 27
Cond 26
Setr 24
Addr 25
Conditions
LireReg 20
EvalueForme 19
Cat 28

12.3. Spécifications de l'analyseur

Etant donné que tout l'environnement dans lequel doit venir s'intégrer l'analyseur est programmé en C, nous avons bien entendu conserver ce même langage-C.

Nous commençons par décrire les structures de données globales [12.3.1.], et les variables globales [12.3.2.] de l'analyseur, avant d'en donner les spécifications proprement dites [12.3.3.].

12.3.1. Structures de données globales

registre : structure chaînée contenant la description d'un registre, à un moment de l'analyse syntaxique [cfr. 10.7.1.].
Il est composé de son nom et de son contenu, et pointe vers le registre suivant.

transition : structure chaînée contenant la description d'un arc lu dans le fichier des ATN.

nom arc : nom de l'arc.

mot : si arc de type CAT, indique la catégorie à laquelle doit appartenir le mot; si arc POP , indique l'état vers lequel on saute; si arc WRD, indique le mot lui-même qui doit être courant.

test : condition sur l'arc.

action : actions associées à l'arc.

liste : si arc de type MEM, indique la liste des mots auxquels le mot courant doit appartenir pour que l'arc soit suivi.

etatsuivant : état suivant vers lequel on saute après exécution des actions.

nextarc : pointeur vers la transition suivante.

stack : structure chaînée qui joue le rôle de pile, pour mémoriser le registre courant et l'arc courant, avant de suivre un arc PUSH (descendre d'un niveau dans les ATN).

regsup : pointeur vers le registre courant.

regstack : pointeur vers l'arc courant.

nextstack : pointeur vers le stack suivant.

configuration : structure chaînée contenant la description d'une configuration, à un moment donné de l'analyse syntaxique [cfr. 12.1.3.].

string : entier qui indique la position dans la phrase analysée.

indicfin : booléen qui indique si on a atteint la fin de la phrase (1 si fin de phrase, 0 sinon).

acregs : pointeur vers le registre courant.

acstack : pointeur vers le stack courant.

lexmode : booléen qui indique le type d'arc qui vient d'être suivi; vaut 0 si arc PUSH, POP, ou JUMP, et dans ce cas, on ne doit pas analyser un nouveau mot; vaut 1 pour les autres types d'arcs, et on doit alors exécuter une nouvelle analyse lexicale de la phrase pour avoir le mot suivant.

acetat : chaîne de caractères qui contient le nom de l'état des ATN dans lequel se trouve la configuration.

acnext : pointeur vers la configuration suivante.

alternative : structure chaînée, qui contient les alternatives possibles à la configuration courante, à un moment donné de l'analyse.

altconfig: pointeur vers la configuration à partir de laquelle l'alternative est calculée, et d'où l'analyseur

repartira s'il n'aboutit pas avec la configuration courante.

alttex : chaîne de caractères contenant le nom du mot courant ou moment à l'alternative est créée.

altnext : pointeur vers l'alternative suivante.

parses : structure chaînée contenant la description d'un résultat de l'analyse syntaxique de la phrase courante.

structure : chaîne de caractères contenant la structure-résultat.

nextparse : pointeur vers le parses suivant.

12.3.2. Variables globales du programme

ATN : fichier contenant les règles grammaticales sous forme d'ATN

Dict : fichier contenant le dictionnaire Esperanto-français.

tab : tableau des suffixes changeant la catégorie grammaticale d'un mot.

pref : tableau des préfixes Esperanto.

accour : pointeur vers la configuration courante, c'est à dire celle en cours d'analyse.

acnewdern : pointeur vers la dernière configuration de la liste chaînée des configurations.
pendant la recherche des successeurs possibles d'une configuration, chaque nouvelle configuration déterminée est placée en fin de la liste et acnewdern pointe vers cette dernière configuration.

altcour : pointeur vers l'alternative courante, dans la liste des alternatives .

regprem : pointeur vers le premier registre de la liste chaînée des registres.

regcour : pointeur vers le registre courant de la liste chaînée des registres.

regdern : pointeur vers le dernier registre de la liste chaînée des registres.

premparse : pointeur vers le premier parses de la liste chaînée des parses (c'est à dire le premier résultat obtenu).

premstack : pointeur vers le premier stack de la liste chaînée des stacks.

lex : chaîne de caractères contenant le mot courant à tout moment de l'analyse.

raclex : chaîne de caractères contenant la racine du mot courant à tout moment de l'analyse.

phrase : chaîne de caractères contenant la phrase courante analysée.

finphrase : (entier) booléen indiquant si on a atteint la fin de la phrase (=1) ou non (=0).

pointeurphrase : entier indiquant l'endroit où on se trouve dans la phrase, c'est à dire le numéro du caractère où on s'est arrêté.

pos : entier indiquant la position courante dans la description d'un mot.

12.3.3. Spécifications des fonctions

1. *Parser*

ARGUMENT : -phrase : chaîne de caractères contenant la phrase Esperanto dont on veut réaliser l'analyse syntaxique.

RESULTAT : -premparse : pointeur vers la structure obtenue par l'exécution de parser.

FONCTION : -Lors de son premier appel, PARSER établit une configuration initiale acfs, avec l'état initial de la grammaire ATN, soit PRINC/.

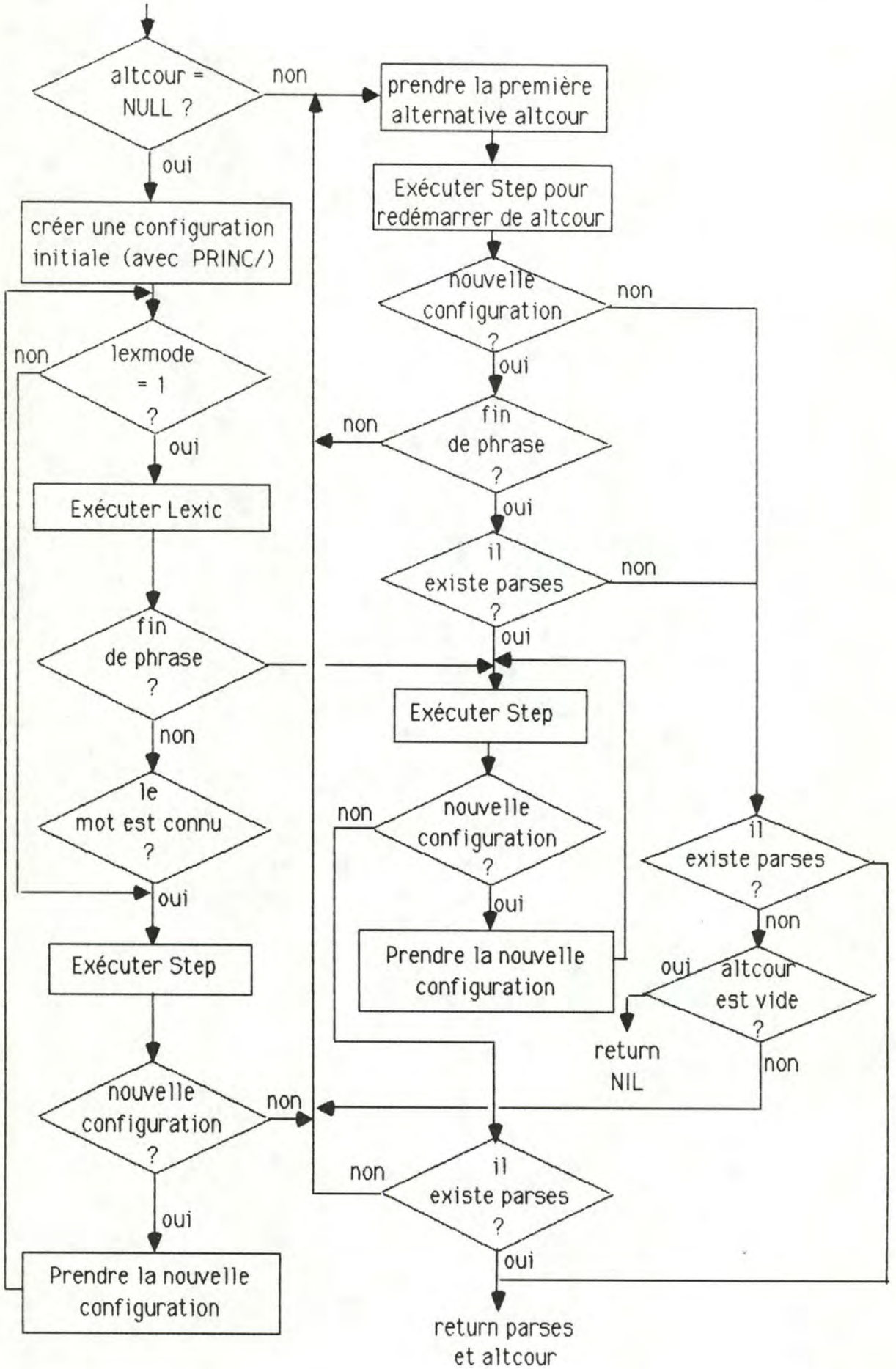
-Elle fait ensuite appel à LEXIC (lorsque le traitement du mot précédent est terminé, c'est à dire lexmode=1), et à STEP, tant que la liste des configurations n'est pas vide.

-Quand la liste des configurations se vide, ou qu'on fait appel à PARSER alors que altcour n'est pas vide, elle prend la première alternative de la liste des alternatives, c'est à dire altcour, et elle exécute STEP afin de reprendre l'analyse à partir de la configuration contenue dans altcour.

-PARSER renvoie 0, quand on est en fin de phrase, que premparse n'est pas vide, et que la configuration courante n'a plus de successeurs.

-PARSER renvoie 1, si elle se termine alors que altcour et premparse sont vides.

Nous présentons à la page suivante l'ordinogramme de la fonction PARSER.


altcour = NULL ?
non
oui
prendre la première alternative altcour
Exécuter Step pour redémarrer de altcour
nouvelle configuration ?
non
oui
fin de phrase ?
non
oui
il existe parses ?
non
oui
créer une configuration initiale (avec PRINC/)
non
lexmode = 1 ?
oui
Exécuter Lexic
fin de phrase ?
non
le mot est connu ?
oui
Exécuter Step
nouvelle configuration ?
non
oui
Prendre la nouvelle configuration
Exécuter Step
nouvelle configuration ?
non
oui
Prendre la nouvelle configuration
il existe parses ?
non
oui
altcour est vide ?
oui
non
return NIL
il existe parses ?
non
oui
return parses et altcour

2. *Lexic*

OBJECTIF : réalise l'analyse lexicale de la phrase, c'est à dire cherche le mot suivant dans la phrase à analyser, et vérifie si il est lexicalement correct.

ARGUMENT : -phrase

PRE : -phrase est une phrase Esperanto que l'on veut analyser.
-pointeurphrase est la position courante dans phrase, c'est à dire l'indice du premier caractère du mot courant.

RESULTATS : -mot
-racine
-0,1, ou 2

POST : -Lexic renvoie 0 si elle a trouvé un mot, dans phrase, à partir de l'indice pointeurphrase; 'mot' contient alors ce mot , racine en est la racine sémantique Esperanto, et pointeurphrase est la position du mot suivant dans phrase.

-Lexic renvoie 1 si mot n'existe pas en Esperanto.

-Lexic renvoie 2 si elle a atteint la fin de phrase; dans ce cas, finphrase vaut 1.

APPELLE : -Morph

3. *Step*

OBJECTIF : calcule la configuration successeur de la configuration donnée et crée des alternatives pour tous les arcs non suivis.

ARGUMENTS : -conf : configuration
-alt : alternative
-type : entier

PRE : -type = 1 ou 2

-si type = 1 alors alt est vide et conf contient la configuration courante dont on veut le successeur.

-si type = 2 alors conf est vide et alt contient l'alternative courante à redémarrer.

RESULTATS : -acint
-altcour

POST : -si type =1, si les conditions sur cet arc sont vérifiées et les actions exécutées correctement, alors acint est la configuration successeur de la configuration conf, pour le premier des arcs associés à conf.

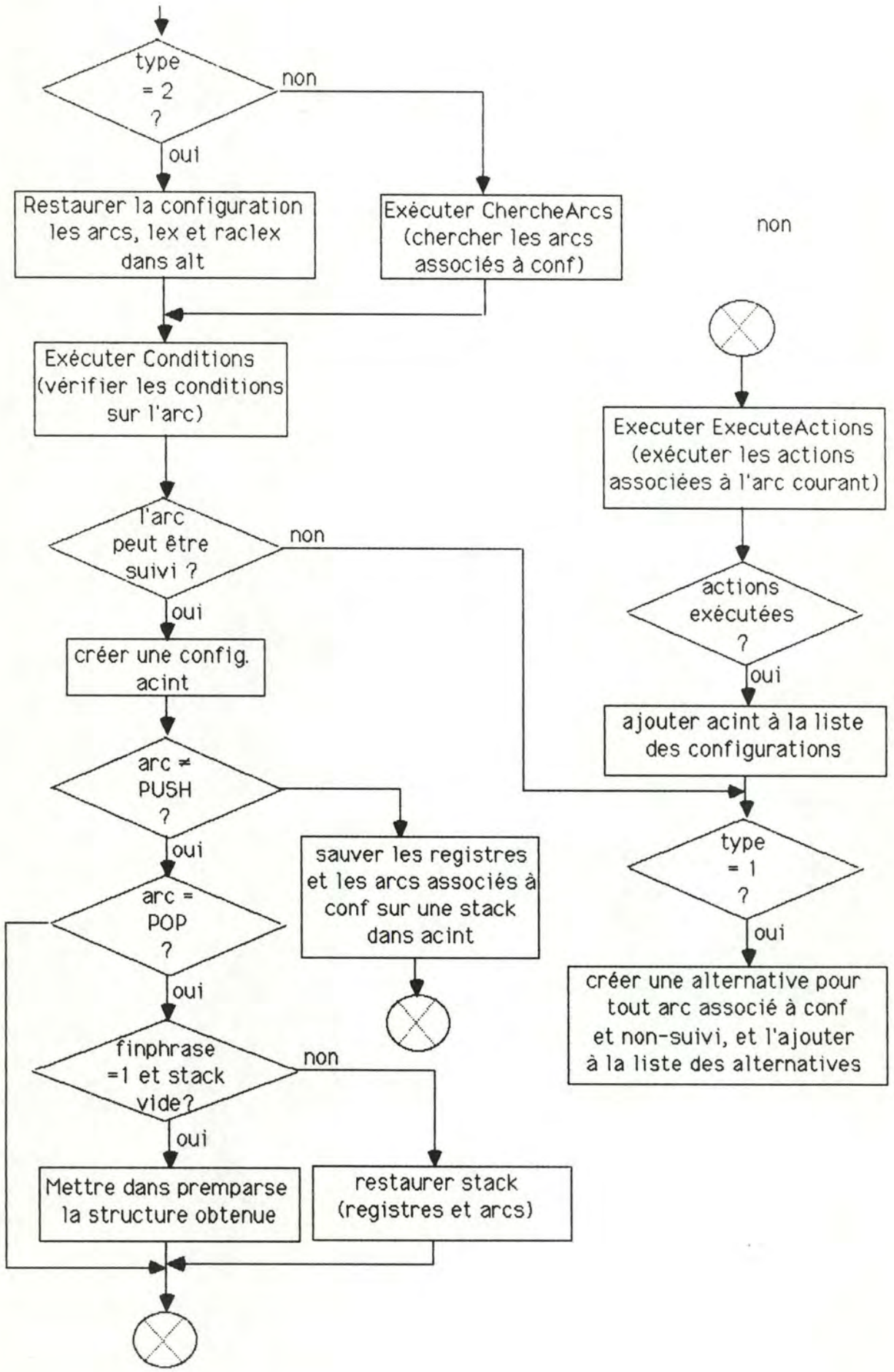
-si type = 2, si les conditions sur cet arc sont vérifiées, et les actions exécutées correctement, alors acint est la configuration successeur de la configuration contenue dans alt.

-acint est ajouté à la fin de la liste des configurations. acnewdern pointe vers acint.

-altcour pointe vers la première des alternatives créée pour tout arc associé à conf et non déjà suivi.

APPELLE: Cat, Mem, Conditions, ChercheArcs , ExecuteActions.

Nous présentons à la page suivante l'ordinogramme de la fonction STEP.


type = 2 ?
non
oui
Restaurer la configuration les arcs, lex et raclex dans alt
Exécuter ChercheArcs (chercher les arcs associés à conf)
Exécuter Conditions (vérifier les conditions sur l'arc)
l'arc peut être suivi ?
non
oui
créer une config. acint
arc ≠ PUSH ?
oui
sauver les registres et les arcs associés à conf sur une stack dans acint
arc = POP ?
oui
finphrase =1 et stack vide?
non
oui
Mettre dans premparse la structure obtenue
restaurer stack (registres et arcs)
non
Executer ExecuteActions (exécuter les actions associées à l'arc courant)
actions exécutées ?
oui
ajouter acint à la liste des configurations
type = 1 ?
oui
créer une alternative pour tout arc associé à conf et non-suivi, et l'ajouter à la liste des alternatives

4. *Morph*

OBJECTIF : réalise l'analyse morphologique d'un mot.

ARGUMENTS : -mot

RESULTATS : -racine
-0 ou 1

POST : -racine est la racine sémantique Esperanto de mot, si mmot est un mot existant en Esperanto.

-racine appartient au dictionnaire Esperanto-français.

-Morph renvoie 0 si 'mot' existe en Esperanto.

-Morph renvoie 1 sinon.

APPELLE : -ChercheMotDict
-LireMotDict
-LireLigneMot

5. *ChercheMotDict*

ARGUMENT : -mot

RESULTAT : -0 ou 1

POST : -ChercheMotDict renvoie 0 si 'mot' existe en Esperanto.

APPELLE : -LireMotSansDescr

6. *LireMotDict*

ARGUMENTS :

RESULTATS : -MotDict
-DescrMotDict
-0 ou 1

POST : -MotDict est une racine sémantique Esperanto.

-DescrMotDict est la rubrique de Dict associée à MotDict.

-LireMotDict renvoie 0 si elle a réussi à lire un mot et sa rubrique dans Dict.

-LireMotDict renvoie 1 si le premier caractère lu est la marque fin de fichier (EOF).

7. *LireLigneMot*

ARGUMENTS : -DescrMot

PRE : -DescrMot est une rubrique complète de Dict, associée à un mot de Dict.

-pos est la position courante dans DescrMot.

RESULTAT : -ligne
-0 ou 1

POST : -ligne est une ligne de DescrMot, c'est à dire la définition d'un mot Esperanto existant à partir d'une racine répertoriée dans Dict.

-pos est positionné au début de la ligne suivante.

-LireLigneMot renvoie 0 si elle a trouvé une ligne.

-LireLigneMot renvoie 1 sinon.

8. *LireMotSansDescr*

ARGUMENTS :

RESULTAT : -MotDict
-0 ou 1

POST : -MotDict est un mot (une racine sémantique Esperanto) contenu dans Dict.

-LireMotSansDescr renvoie 0 quand elle a terminé de lire un mot dans Dict.

-LireMotSansDescr renvoie 1 si le premier caractère lu est la marque fin de fichier (EOF).

9. *ChercheArcs*

ARGUMENT : -eta

PRE : -eta contient le nom d'un état de la grammaire ATN, c'est à dire appartenant au fichier ATN.

RESULTAT : -arc

POST : -arc pointe vers le premier des arcs quittant eta (ceux-ci sont chaînés).

APPELLE : -LireEtat
-LireArc

10. *LireEtat*

ARGUMENT :

RESULTATS : - 0 ou 1
-etatlu
-descretatlu

POST : -LireEtat renvoie 0 si elle a lu un état et sa description dans ATN. etatlu contient alors le nom de l'état lu dans ATN et descretatlu contient la description de étatlu , c'est à dire la description des arcs associés à etatlu dans ATN.

11. *LireArc*

ARGUMENT : -descretat

PRE : -descretat est la description d'un état de ATN.

-i est la position de l'arc courant dans descretat.

RESULTATS : -0 ou 1
-arc : transition

POST : -LireArc renvoie 0 si elle est parvenue à lire un arc entièrement; arc pointe alors vers l'arc créé qui contient la description de l'arc lu dans descretat à partir de la position i Les différents champs de arc sont remplis. [cfr 12.3.1.]

12. *Conditions*

ARGUMENT : -test

PRE : -test contient la description d'une condition associée à un arc de ATN.

RESULTAT : 0 ou 1

POST : -Conditions renvoie 0 si la condition test est vérifiée.

-Conditions renvoie 1 sinon

APPELLE : -Cond_Or, Cond_Nullr, Cond_Equal, Cond_And, Cond_Appartient, Cond_Start.

RESULTAT : 0 ou 1

POST : -Cond_Or renvoie 0 si la condition est vérifiée, c'est à dire si cond 1 **ou** cond 2 est vérifiée

13. *Cond_Or*

ARGUMENT : -test

PRE : -test est une condition sur un arc de ATN, de type "OR" : (OR (cond 1) (cond 2)).

RESULTAT : -0 ou 1

POST : -Cond_Or renvoie 0 si la condition est vérifiée, c'est à dire si cond 1 **ou** cond 2 est vérifiée.

-Cond_Or renvoie 1 sinon.

APPELLE : -Conditions

14. *Cond_And*

ARGUMENT : -test

PRE : -test est une condition sur un arc de ATN, de type "AND": (AND (cond 1) (cond 2)).

RESULTAT : -0 ou 1

POST : -Cond_And renvoie 0 si la condition est vérifiée, c'est à dire si cond 1 **et** cond 2 sont vérifiées.

-Cond_And renvoie 1 sinon.

APPELLE : -Conditions

15. *Cond_Equal*

ARGUMENT : -test

PRE : -test est une condition sur un arc de ATN, de type "EQUAL" : (EQUAL (forme 1) (forme 2)).

RESULTAT : -0 ou 1

POST : -Cond_Equal renvoie 0 si la condition est vérifiée, c'est à dire si les valeurs de forme 1 et de forme 2 sont équivalentes.

-Cond_Equal renvoie 1 sinon.

APPELLE : -EvalueForme

16. *Cond_Nullr*

ARGUMENT : -test

PRE : -test est une condition sur un arc de ATN, de type "NULLR" : (NULLR reg)

RESULTAT : -0 ou 1

POST : -Cond_Nullr renvoie 0 si la condition est vérifiée, c'est à dire si le registre 'reg' a une valeur nulle, ou s'il n'a jamais eu de valeur.

-Cond_Nullr renvoie 1 sinon.

APPELLE : -LireReg

17. *Cond_Appartient*

ARGUMENT : -test

PRE : -test est une condition sur un arc de ATN, de type "APPARTIENT" ; (APPARTIENT (forme)(liste))

RESULTAT : -0 ou 1

POST : -Cond_Appartient renvoie 0 si la condition est vérifiée, c'est à dire si la valeur de 'forme' se retrouve dans la liste (contenue dans test).

-Cond_Appartient renvoie 1 sinon.

APPELLE : -EvalueForme

18. *Cond_Start*

ARGUMENT : -test

PRE : -test est une condition sur un arc JUMP ou PUSH de ATN, de type "-START"; ('type'-START).

RESULTAT : -0 ou 1

POST : -Cond_Start renvoie 0 si la condition est vérifiée, c'est à dire si lex est de catégorie grammaticale 'type'.

-Cond_Start renvoie 1 sinon.

APPELLE : -Cat

19. *EvalueForme*

ARGUMENT : -forme

PRE : -forme est une forme [cfr. 10.5.4.] sur un arc de ATN.

RESULTAT : -eval

POST : -eval contient le résultat de l'évaluation de forme.

-si forme de type 'LEX' ou '*', eval contient la valeur de lex.

-si forme de type 'QUOTE', eval contient ce qui suit QUOTE.

-si forme de type 'GETR', eval contient la valeur du registre spécifié après GETR.

-si forme de type 'BUILDQ', eval contient la structure construite à partir des registres [cfr 10.4.]

-si forme de type 'NIL' ou 'T', eval contient 'NIL' ou 'T'.

20. *LireReg*

ARGUMENT : -nom

RESULTAT : -val
-0 ou 1

POST : -LireReg renvoie 0 si le registre de nom 'nom' existe; val prend alors sa valeur.

-LireReg renvoie 1 si le registre de nom 'nom' n'existe pas.

21. *ExecuteActions*

ARGUMENT : -arc

PRE : -arc est un pointeur vers un arc contenu dans ATN.

RESULTAT : -0 ou 1

POST : -ExecuteActions renvoie 0 si les actions contenues dans arc ont pu s'exécuter correctement.

-ExecuteActions renvoie 1 sinon.

APPELLE : -ExecuteUneAction

22. ExecuteUneAction

ARGUMENT : -action
-arc

PRE : -action est une action contenue dans arc.

RESULTAT : -0 ou 1

POST : -ExecuteUneAction renvoie 0 si l'action 'action' s'est exécutée correctement.

-ExecuteUneAction renvoie 1 sinon.

APPELLE : -To, Setr, Addr, Cond, Verify.

23. To

ARGUMENT : -action
-arc

PRE : -action est une action contenue dans arc (arc de ATN), et est du type (To <etat suivant>).

RESULTAT : -arc->etatsuivant

POST : -arc->etatsuivant contient le nom de l'état indiqué après 'To, dans action.

24. Setr

ARGUMENT : -action

PRE : -action est une action contenue dans un arc de ATN et est du type (SETR <registre> <forme>).

RESULTAT : -regcour

POST : -regcour pointe vers le nouveau registre créé , de nom 'registre' , qui contient la valeur de forme, et qui est ajouté au début de la liste des registres.

APPELLE : -EvalueForme

25. *Addr*

ARGUMENT : -action

PRE : -action est une action contenue dans un arc de ATN, et est de type (ADDR <registre><forme>).

RESULTAT : -regcour

POST : -regcour pointe vers le nouveau registre créé, de nom 'registre', qui contient la valeur de 'registre à laquelle on a ajouté la valeur de la forme et qui est ajouté au début de la liste des registres.

APPELLE : -EvalueForme
-LireReg

26. *Cond*

ARGUMENT : -action
-arc

PRE : -action est une action contenue dans arc (arc de ATN), et est du type (COND (cond 1) (action 1)

...

(cond n) (action n))

RESULTAT : -0 ou 1

POST : -Cond renvoie 0 si une des conditions cond i est vraie, et si l'action qui lui correspond s'est exécutée correctement.

-Cond renvoie 1 sinon.

APPELLE : -Conditions
-ExecuteUneAction

27. *Verify*

ARGUMENT : -action

PRE : -action est une action contenue dans un arc de ATN, et est du type (VERIFY (cond))

RESULTAT : -0 ou 1

POST : -Verify renvoie 0 si la condition 'cond' est vérifiée.

-Verify renvoie 1 sinon.

APPELLE : -Conditions

28. *Cat*

ARGUMENTS : -mot
-testmot
-racinemot

PRE : -testmot est un mot de la phrase à analyser.

-racinemot est la racine sémantique Esperanto de 'testmo't.

-mot est une catégorie grammaticale; mot appartient (subjonction, article, adjectif, préposition, adverbe, pronom, numéral, abréviation, verbe, substantif).

RESULTAT : -0 ou 1

POST : -Cat renvoie 0 si 'testmot' est de la catégorie grammaticale 'mot'.

-Cat renvoie 1 sinon.

29. *Mem*

ARGUMENT : -liste (liste de mots)

RESULTAT : -0 ou 1

POST : -Mem renvoie 0 si lex appartient à la liste.

-Mem renvoie 1 sinon.

12.3.4. Format de la structure-résultat

Le résultat de l'analyse syntaxique (la structure syntaxique de la phrase en entrée) doit uniquement servir comme donnée du programme de génération du texte français.

Cependant, afin de pouvoir vérifier rapidement le résultat fourni par l'analyseur, nous avons choisi de présenter ce résultat sous une forme compréhensible par une personne humaine.

Ce résultat se présente sous la forme d'une liste d'éléments entre parenthèses, qui commence toujours par "PRINC" pour signaler le début de la structure d'une phrase.

Rappelons tout d'abord qu'une phrase a la forme linéaire suivante :

PRINC = VAL1 | VERBE | VAL2 | ATTR | VAL3 | CIRC

Elle est donc composée de six fonctions syntaxiques (facultatives) qui sont elles-mêmes représentées, dans la structure-résultat, par une liste du même type que cette structure-résultat. En effet, ces structures commencent par le nom de la fonction syntaxique, et contiennent ensuite les différents syntagmes, et les catégories syntaxiques qui les composent. Chaque syntagme est lui-même représenté sous la forme d'une liste entre parenthèses formée à partir des catégories syntaxiques qui le constituent; chaque catégorie dont un mot est présent dans la phrase, pour le syntagme et la fonction correspondants, est inscrit dans la structure après le nom de la catégorie.

Voici un exemple de structure-résultat correspondant à la phrase "La BelA InfanO EstAs BonA" :

(PRINC (VAL1 (GN (DET1 (GADJ (DET1 ((ART La))) (ADJ BelA) (CIRC)) (N InfanO) (DET2) (COMP) (CIRC))) (V EstAs) (VAL2) (ATTR (GADJ (DET1) (ADJ BonA)(CIRC))) (VAL3) (CIRC))

où "La BelA InfanO" est un syntagme nominal qui joue le rôle de une VALENCE1 (VAL1) par rapport au verbe "EstAs"; en outre, "La BelA" est un groupe adjectival (GADJ) qui joue le rôle de DETERMINANT1 (DET1) pour le substantif "InfanO", et est constitué de l'article "La" (ART) , de l'adjectif "BelA" (ADJ), et ne contient pas de circonstants (CIRC est suivi d'un blanc). La phrase ne contient ni de VALENCE2, ni de VALENCE3, ni de CIRCONSTANTS. Elle contient un ATTRIBUT du verbe : "BonA" qui est un adjectif appartenant à un groupe adjectival...

Chapitre 13 : la génération du texte français

13.1. Introduction

La dernière étape du système de traduction dont nous avons commencé la réalisation, consiste à générer une phrase correcte en français, à partir de la structure syntaxique de la phrase Esperanto obtenue à l'aide de l'analyseur syntaxique présenté au chapitre 12.

Cette étape implique en premier lieu une transposition de structure qui consiste à convertir la structure syntaxique Esperanto obtenue en structure syntaxique française; cette conversion de structure est nécessaire car les structures syntaxiques des phrases dans des langues différentes ne sont pas toujours équivalentes (elles le sont même rarement). Nous appellerons ce changement de structure **métataxe**, par respect pour Tesnière.

Pour réaliser la métataxe, dans notre système, il est nécessaire de disposer d'un dictionnaire Esperanto-français plus puissant que celui que nous avons construit jusqu'à présent [cfr.chap.8].

C'est pourquoi nous avons améliorer ce dictionnaire en y incorporant tout un ensemble d'informations indispensables pour une "bonne" métataxe [cfr. 13.2.].

En second lieu, il reste à convertir la structure syntaxique française obtenue en ordre linéaire, c'est à dire à écrire la phrase française qui correspond à cette structure.

Nous n'avons malheureusement pas eu le temps d'implémenter le programme qui réalise cette génération en français, mais nous exposons cependant en 13.3. le principe sous-jacent à cette dernière étape de notre système de traduction, à l'aide d'un exemple concret.

13.2. Présentation du dictionnaire

La forme du dictionnaire est identique à celle présentée au chapitre 8, excepté pour ce qui est de la partie "traductions" en français, c'est à dire tout ce qui suit le signe "=" dans une ligne du dictionnaire.

Nous présentons ci-dessous les modifications apportées, en fonction de la catégorie syntaxique à laquelle le mot appartient. Nous supposons que toute ligne du dictionnaire se décompose en colonnes, et nous exposons par conséquent les colonnes qui ont été ajoutées, après la colonne "traductions".

13.2.1. Pour le verbe

A) Le verbe, dans l'acception concernée, s'utilise-t-il pronominalement ou non ? : O(ui) | N(on). En Esperanto, le verbe est "pronominalisé" en lui intégrant un suffixe, contrairement au français, ex : Nomi**Gh**i = **s'**appeler.

B) Le verbe peut-il avoir un prime-actant ? : O | N. Si c'est non, il s'agit d'un verbe impersonnel.

C) Le verbe peut-il avoir un second-actant ? : O | N.

D) Le verbe peut-il avoir un tiers-actant ? : O | N.

E) A quelle conjugaison le verbe appartient-il ? : numéro qui renvoie à une série de tableaux de conjugaisons du verbe, repris du livre de Bertrand [BERTR], qui en recense une centaine.

F) Avec quel auxiliaire le verbe se conjugue-t-il dans cette acception : A(voir) | E(tre). (Ex : il a monté le paquet; il est monté dans le train.)

13.2.2. Pour les noms

A) Quel est leur genre ? : F(éminin) | M'asculin).

B) Quel est leur pluriel ? : numéro qui renvoie à une série de six règles de formation du pluriel.

C) Nous ajouterons peut-être une colonne "sémantique" pour déterminer à quel champ lexical le nom appartient : médecine, astrologie, ...

13.3.3. Pour les adjectifs

A) Comment se forme leur féminin ? : numéro qui renvoie à une série de treize règles.

B) et C) idem que pour les noms.

N.B : Nous n'avons pas encore introduit les mots-outils.

13.3. Exemple de génération d'une phrase française

Reprenons la phrase "La BelA InfanO EstAs BonA" pour laquelle nous avions obtenue, à l'aide de notre analyseur syntaxique (au paragraphe 12.3.4.), la structure syntaxique suivante :

(PRINC (VAL1 (GN (DET1 (GADJ (DET1 ((ART La))) (ADJ BelA) (CIRC)) (N InfanO) (DET2) (COMP) (CIRC))) (V EstAs) (VAL2) (ATTR (GADJ (DET1) (ADJ BonA)(CIRC))) (VAL3) (CIRC))

Nous parcourons cette structure, et pour chaque mot rencontré, nous allons chercher sa ou ses traductions dans le dictionnaire, ainsi que toutes les informations nécessaires.

Nous rencontrons d'abord l'article (ART) "La"; le dictionnaire nous dit :

La -> le, la, l', les.

A ce moment de l'analyse, rien ne permet de dire si la traduction correcte est "le", "la", "l'", ou "les"; il faut attendre de rencontrer le nom dont dépend l'article.

Nous passons au mot suivant, l'adjectif (ADJ) "BelA", pour lequel on obtient:

BelA -> beau, bel, joli.

Une routine sémantique permettra de sélectionner "beau" ou "bel", plutôt que "joli", et sachant que l'adjectif s'accorde en genre et en nombre avec son régissant, qui est le nom (ou substantif), il faut attendre d'atteindre celui-ci pour l'accord de l'adjectif.

Le mot suivant est le nom "InfanO", pour lequel le dictionnaire nous donne la traduction "enfant", et le genre "masculin". Nous savons également que InfanO est au singulier (pas de marque du pluriel "J").

Nous pouvons maintenant revenir en arrière pour accorder l'article et l'adjectif; étant donné que le nom est au masculin singulier, l'article et l'adjectif le sont également.

Pour l'article, nous pouvons donc éliminer "la" et "les"; il reste à choisir entre "le" et "l'", en fonction de la première lettre du mot qui suit l'article. Comme il s'agit de l'adjectif (une règle nous dit que l'adjectif se place avant le nom), et que celui commence par une consonne ("b"), l'article sera donc "le".

Le nom étant au masculin singulier, l'adjectif le sera également; en outre, l'adjectif précédant directement le nom, et celui-ci commençant par une voyelle ("Enfant"), l'adjectif sera "bel" et non "beau".

A ce stade, nous avons donc la structure du groupe nominal français :

(GN (ART le) (ADJ bel) (N enfant))

Le mot suivant de la structure Esperanto est le verbe "EstAs", qui est le régissant principal de la phrase; le dictionnaire nous dit qu'il s'agit du verbe "EstI" de valence 1 (V1), dont la traduction est "être", également de valence 1. Une règle nous dit que le verbe être s'accorde en personne avec son sujet, et celui-ci ("enfant") est au singulier; de plus, le verbe est au présent (terminaison "As"). En outre, on sait que dans les répertoires nous n'avons jamais que la troisième personne, ce qui fait que la seule traduction possible pour le verbe est "est" (troisième personne du singulier du verbe "être").

Nous obtenons :

((GN (ART le) (ADJ bel) (N enfant)) (V est))

Le dernier mot de la phrase est l'adjectif attribut (ATTR) "BonA", qui signifie "bon", "gentil" (supposons qu'une routine sémantique est sélectionné "gentil").

Comme un attribut s'accorde avec le sujet, "gentil" s'accorde avec "enfant", et est donc au masculin singulier : "gentil".

La structure syntaxique de la phrase, en français, est donc :

((VAL1 (GN (ART le) (ADJ bel) (N enfant))) (V est) (ATTR (ADJ gentil)))

La dernière étape consiste à transposer cette structure en ordre linéaire, à l'aide de règles de distribution en français.

Nous obtenons la phrase :

"Le bel enfant est gentil".

CONCLUSION

Ce mémoire avait pour but de contribuer à la réalisation d'un système de traduction (semi-)automatique des répertoires homéopathiques du système RADAR.

Il faisait suite à un mémoire déja réalisé l'année dernière [HOGNE], et le programme de traduction devait par conséquent venir s'intégrer dans l'environnement créé par nos prédecesseurs; nous avons opté, comme ils le conseillaient, pour la technique de traduction via un interlangage, en prenant l'Esperanto comme langue intermédiaire.

Nous avons choisi, pour des raisons évidentes de facilité, le français comme première langue-cible de notre système.

La traduction comportait deux étapes bien distinctes :

-une étape de traduction anglais-Esperanto.
-une étape de traduction Esperanto-français.

Pour la réalisation de la première étape, nous avons préféré une traduction "manuelle" des répertoires, et nous avons créé divers outils informatiques (dont un dictionnaire automatisé anglais-Esperanto et les primitives d'accés à celui-ci), qui ont grandement simplifié la tâche du traducteur, et ont permis d'obtenir une version Esperanto des répertoires de qualité.

En ce qui concerne la deuxième étape, nous avons encodé un dictionnaire Esperanto-français très complèt, et nous avons écrit une grammaire Esperanto en utilisant la grammaire de dépendance de Tesnière; nous avons ensuite formalisé cette grammaire en employant la technique des "Augmented Transition Networks". Enfin, nous sommes parvenus à construire un analyseur syntaxique, qui fournit la structure syntaxique de toute phrase Esperanto rencontrée dans les répertoires, sur base des règles de grammaire (les ATN), et du dictionnaire Esperanto-français.

Il reste maintenant à réaliser le programme de génération d'une phrase française à partir de la structure Esperanto de cette phrase; le dictionnaire Esperanto-français contient déja pratiquement toute l'information nécessaire à ce travail.

Bibliographie

[ALLEN] : Allen, T.F.A : "The encyclopedia of Pure Materia Medica", B. Jain Publ., New-Delhi, 1977, (12 volumes).

[BATES] : Bates, M. : " The theory and practice of ATN grammars " (Natural Language Communication with Computers- Leonard Bolc - pp. 191-259).

[BERTR] : BERTRAND : "Dictionnaire pratique desq conjugaisons", Pluriguide, Nathan.

[BURTON] : Burton, RR. & Woods, W.A. : "A compiling system for ATN" (International Conference on Computational Linguistics, Ottawa, Canada, juin 1976).

[CHOMSKY] : Chomsky, N. : "Structures Syntaxiques", Seuil, Paris, 1969.

[EARLEY] : Earley, J. : "An efficient Context-Free Parsing Algorithm", Communication of the ACM, 13, 1970 (pp. 94-102).

[EURO] : Ensemble d'articles sur le système Eurodicautom. Revue TERMINOLOGIE, 1981, n° 38-40.

[HAHNEM] : Hahnemann, S. : "Organon of medicine", Indian Books, New-Delhi.

[HERING] : Hering, M.D. : "The guiding symptoms of Pure Materia Medica", B. Jain Publ., New-Delhi, 1971, (10 volumes).

[HOGNE] : Hogne, J-P. : "Traduction automatique de répertoires homéopathiques" Mémoire Institut d'informatique-1986-.

[KENT] : Kent, J.T. : "Repertory of the homeopathic Materia Medica ".

[LAVOREL] : Lavorel, B. : "La traduction automatique à la commission des communautés européennes", article de la revue LE LINGUISTE, n° 4-5, 1983, (pp. 5-9).

[LEROY] : Leroy, M. : "Les Grands Courants de la Linguistique Moderne", Ed de l'Université de Bruxelles, 1971.

[LUNAR] : Woods, W.A. & Kaplan, R.M. & Nash-Weber, B. : "The Lunar Sciences Natural Language Inforamation System : Final Report", BBN Report n° 2378, Bolt Berenek and Newman Inc., Cambridge 1972.

[NIDA] : Nida, E. : "Language structure and translation" (Stanford University Press, 1975).

[PSYCHO] : Kaplan, R-M. : "Augmented Transition Networks as psychological models of Sentence comprehension" (Artificial. Intelligence 3 - pp. 77-100) (1972).

[RADAR] : Fichefet, J. & Jacques, A. & Paris, J. : "R.A.D.A.R : un système expert à base de connaissances pour l'homéopathie"

[RUWET] : Ruwet, N. : "Introduction à la grammaire générative", Plon 1968.

[SCHUBERT] : Schubert, K. : "Syntactic analysis of DLT intermediate language" (BSO - DLT).

[SEMANT] : "Problèmes de Sémantique" (en collaboration) , Presses de l'Université du Québec, 1973.

[TESNIERE] : Tesnière, L. : "Eléments de Syntaxe structurale", Klincksiek, Paris, 1969.

[WITKH] : Witkham : " Feasibility study of a Multilingual Facility for Videotext Information Networks" (BSO- DLT-).

[WOODS1] : Woods, W.A. : "Transition networks Grammar for Natural Language analysis" (Communication of ACM 13 [10] - pp 591-606) (1970)

[WOODS2] : Woods, W.A. : "An experimental parsing system for transition network grammars", Bolt, Beranek and Newman, Inc. (1973).

Autres références consultées :

Aho-Ullman : "The theory of parsing, translation and compiling " (Vol.1 - Parsing Prentice Hall).

Barr, A. & Feigenbaum, E. : "The handbook of Artificial Intelligence" (Vol.1 - Pitman).

Barthel, H. & Klunker, W.H. : "Synthetic repertory (3 vol.).

Breckx, M. : "Initiation à la Linguistique et à la Grammaire Nouvelle", Deboeck, Bruxelles, 1977.

Charniak, E. : " A parser with something for everyone " (King, M-Parsing Natural Language - (p. 117-149) - 1983).

Charniak, E. & MacDermott, D. : "Introduction to Artificial Intelligence" (Addison Wesley Publish. Company).

De Roeck : "An underview of Parsing " (King. p3-17).

Ducrot, O. & Todorov, T. : "Dictionnaire Encyclopédique des Sciences du Langage", Paris, 1972.

Grévisse, M. : "Le bon usage", Grammaire Française, Duculot, S.A., Gembloux, 1964.

Jacques, A. : "Introduction à l'homéopathie hahnemanienne", UIHN, Namur, 1983.

Johnson, R. : "Parsing with transition networks " (King. - p. 59-72).

Kerningham & Ritchie : "Le langage C" (Masson, Paris 1983).

Léger, R. & Albault, A. : "Dictionnaire Français-Esperanto", Ed. Françaises d'Esperanto, Marmande, 1961.

"Le Langage", Dictionnaire, ed. B. Pottier, Paris, 1973.

Leroy, H. : "Cours de théorie des langages" (FNDP-Institut d'informatique).

Pereira, F.C. & Warren, D. : "Definite clause grammars for Language Analysis- A Survey of the formalism and a comparaison with ATN" (Artific. Intelligence 13 - pp. 231- 278).

"Plena Analiza Gramatiko", UEA, Rotterdam, 1985.

"Plena Ilustrita Vortaro de Esperanto", SAT, Paris, 1981.

"Praktiku Kun ni esperanton !" (Esperantista Grupo Universitata de Bonn).

Sampson : "Deterministic Parsing " (King - pp. 91-116) .

"Teach Yourself Dictionary", Esperanto-English, ed. JC Wells, Hotter and Stoughton, 1969.

Van Lamsweerde, A. : "Méthodologie de développement de logiciels", cours Institut d'informatique, FNDP, Namur.

Waringhien, G. : "Grand Dictionnaire Esperanto-Français", SAT Amikaro, Paris, 1976.

Woods, W.A. : "Cascaded ATN grammars" (American Journal of Computational Linguistics, 6 - n°1 - p. 1-12).

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX
NAMUR

INSTITUT D'INFORMATIQUE

TRADUCTION (SEMI-)AUTOMATIQUE

DES REPERTOIRES DU SYSTEME RADAR.

-ANNEXES-

Promoteur : J. FICHEFET.

Mémoire présenté par
D. Jamme
en vue de l'obtention du
titre de Licencié et
Maître en Informatique.

Année académique 1986-1987

ANNEXES:

ANNEXE 1 : Les arbres syntaxiques

ANNEXE 2 : Les ATN sous forme de graphes

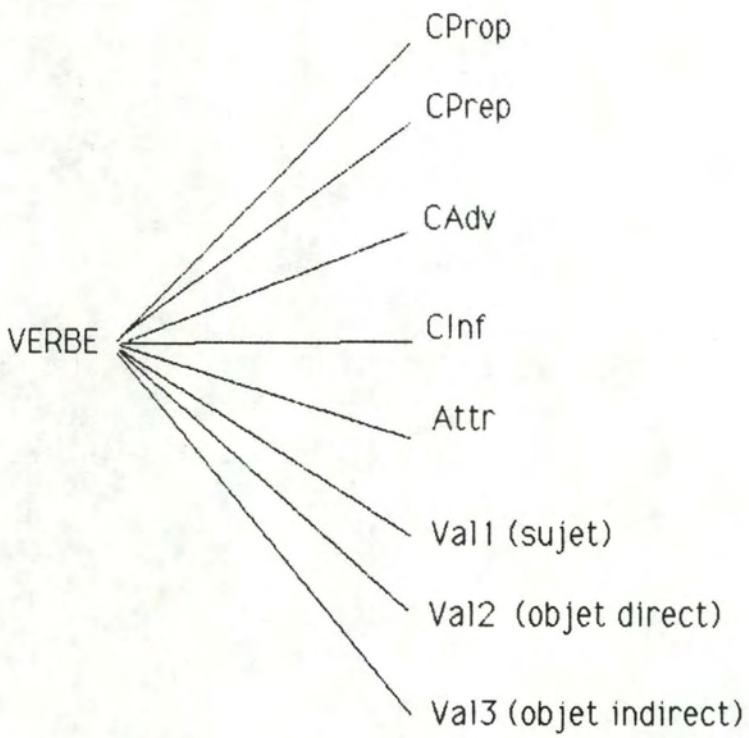
ANNEXE 3 : Le fichier des ATN

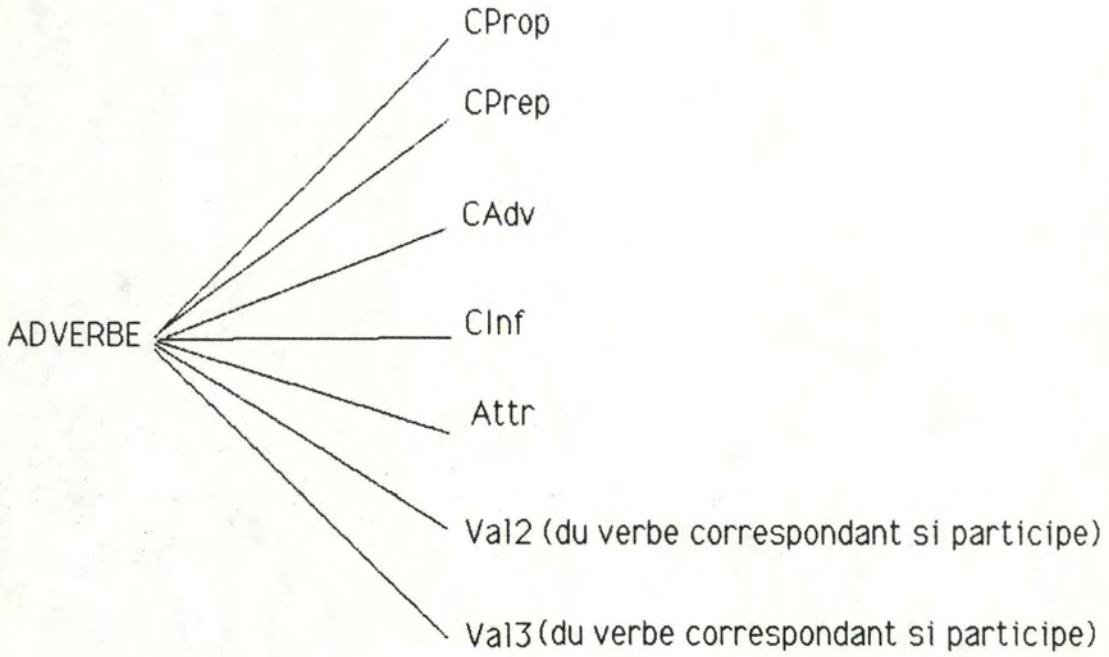
ANNEXE 4 : Les programmes de traduction anglais-Esperanto

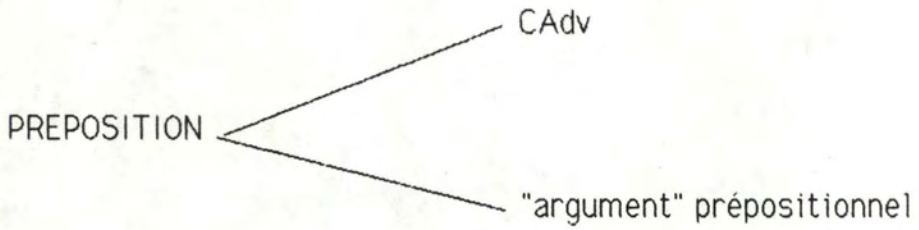
ANNEXE 5 : Les fonctions-C de l'analyseur syntaxique

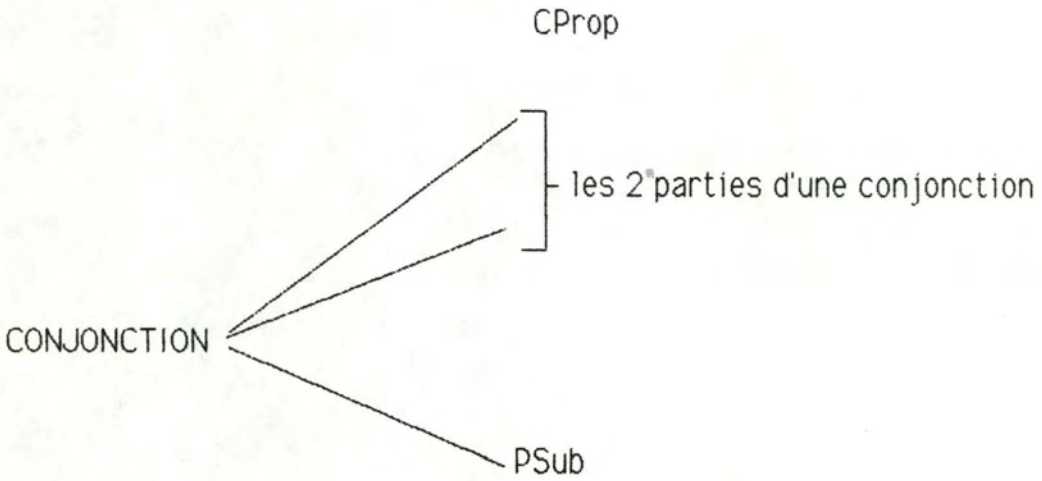
Annexe 1 :

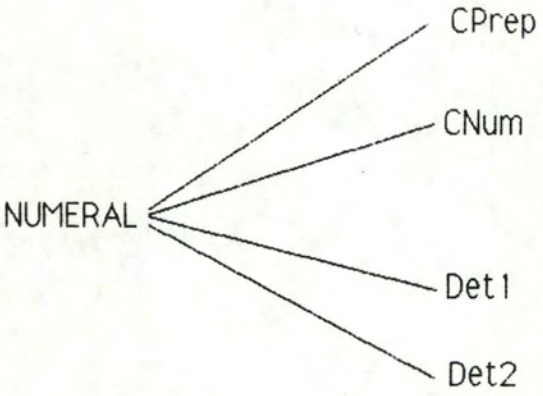
Les arbres de dépendance

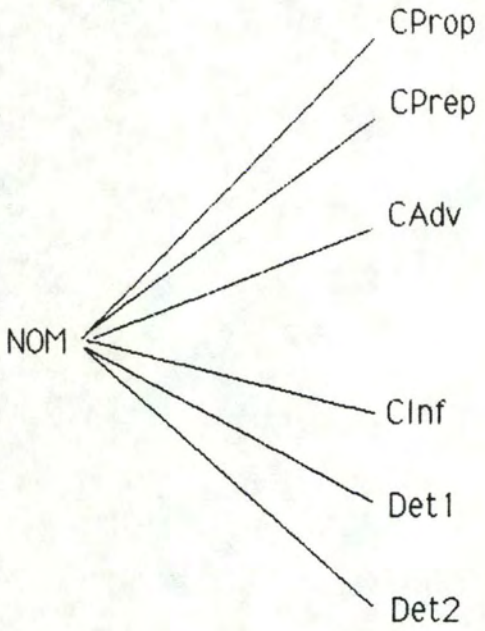

VERBE
CProp
CPrep
CAdv
CInf
Attr
Val1 (sujet)
Val2 (objet direct)
Val3 (objet indirect)

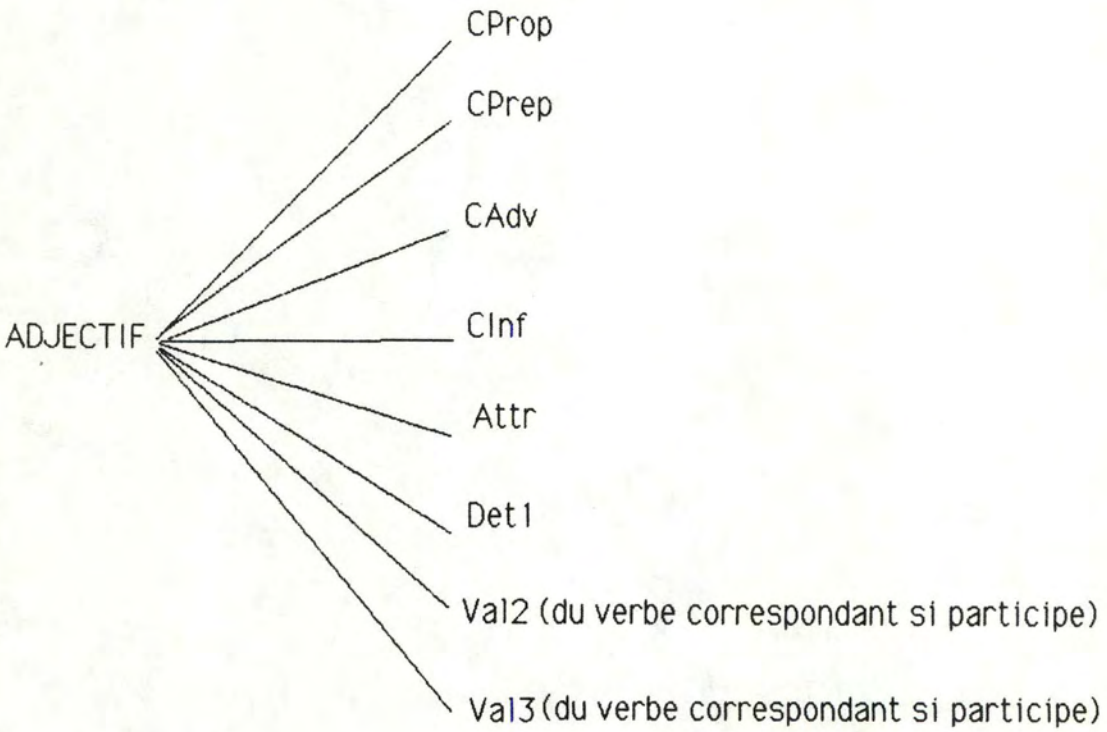

ADVERBE
CProp
CPrep
CAdv
CInf
Attr
Val2 (du verbe correspondant si participe)
Val3 (du verbe correspondant si participe)

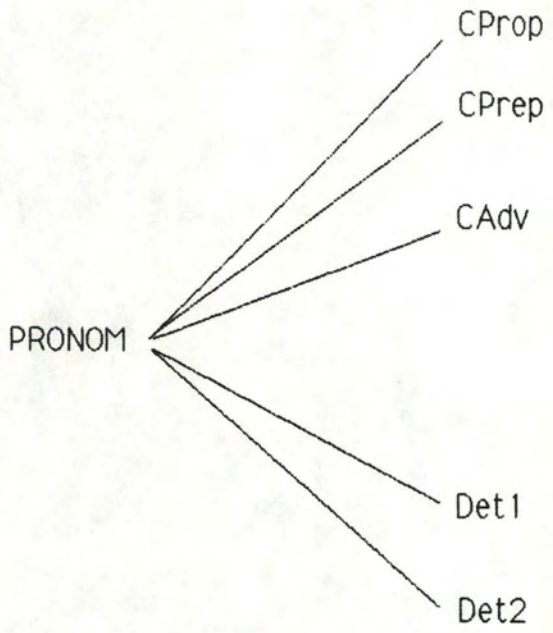

CAdv
PREPOSITION
"argument" prépositionnel


CProp
les 2 parties d'une conjonction
CONJONCTION
PSub


CPrep
CNum
NUMERAL
Det1
Det2


CProp
CPrep
CAdv
NOM
CInf
Det1
Det2


CProp
CPrep
CAdv
ADJECTIF
CInf
Attr
Det1
Val2 (du verbe correspondant si participe)
Val3 (du verbe correspondant si participe)


CProp
CPrep
CAdv
PRONOM
Det1
Det2

Annexe 2 :

Les ATN sous forme de graphes

Légende :

xyz : CAT xyz.

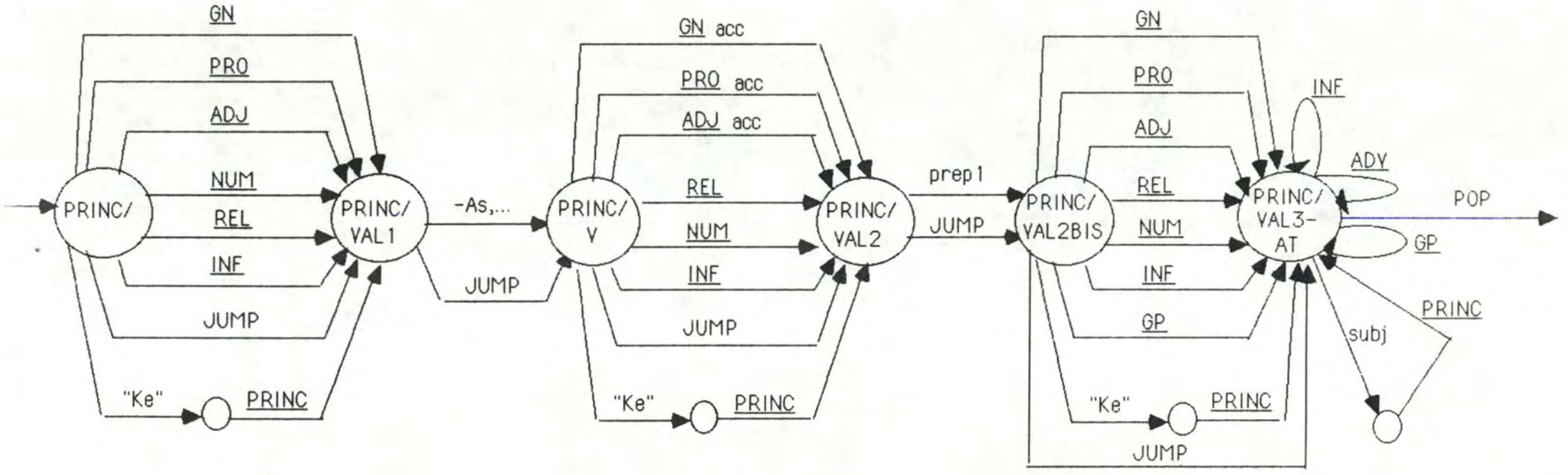
XYZ : PUSH XYZ/ (saut au diagramme XYZ/).

XYZ_{liste} : PUSH XYZ/ si le mot courant appartient à liste.

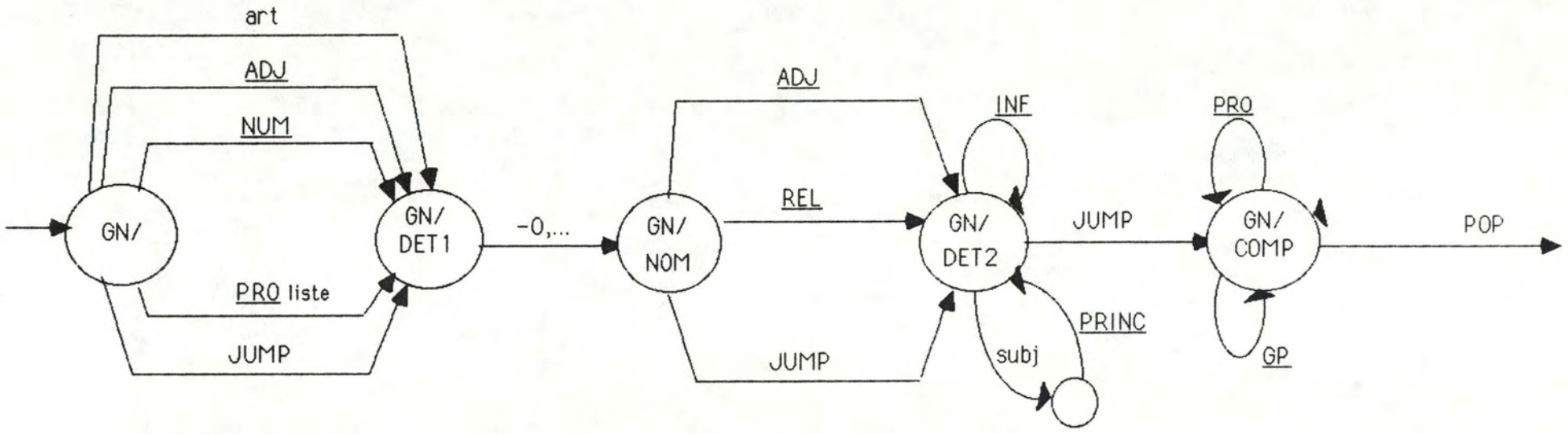
-As : indication de la terminaison grammaticale du mot courant
(-As=présent, I=infinitif, ...).

"xyz" : WRD xyz (le mot courant doit être le mot "xyz").

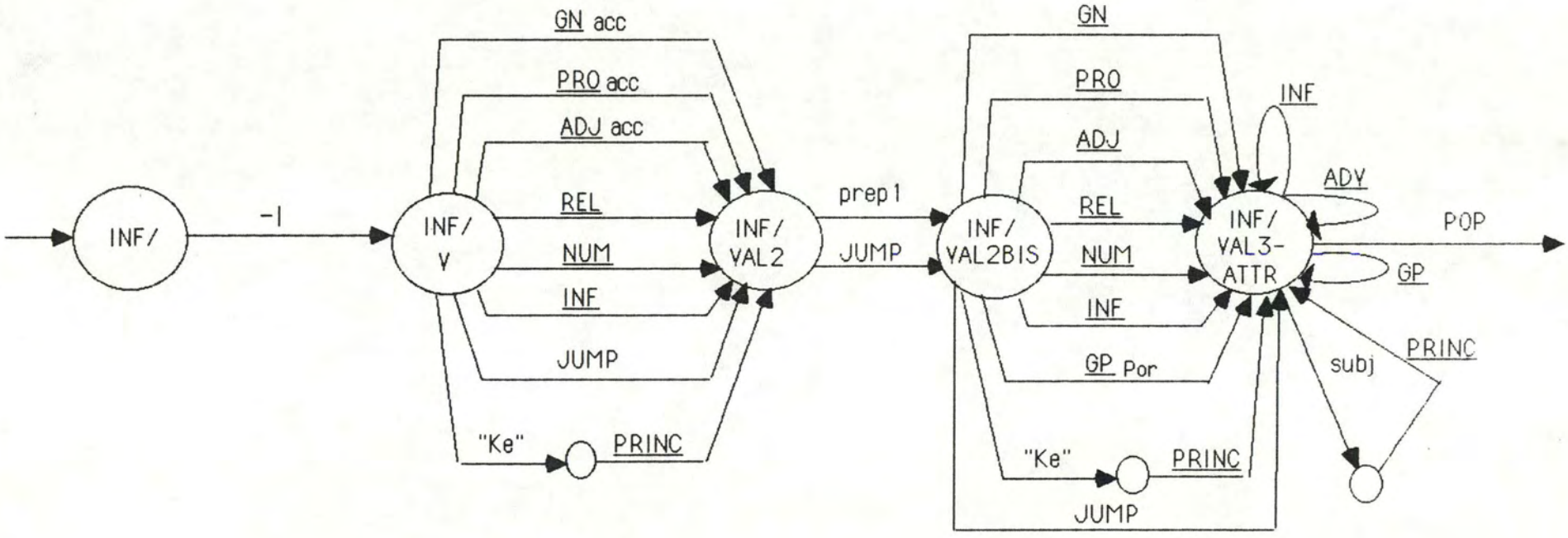
("xyz",...) : MEM liste ("xyz", ...).

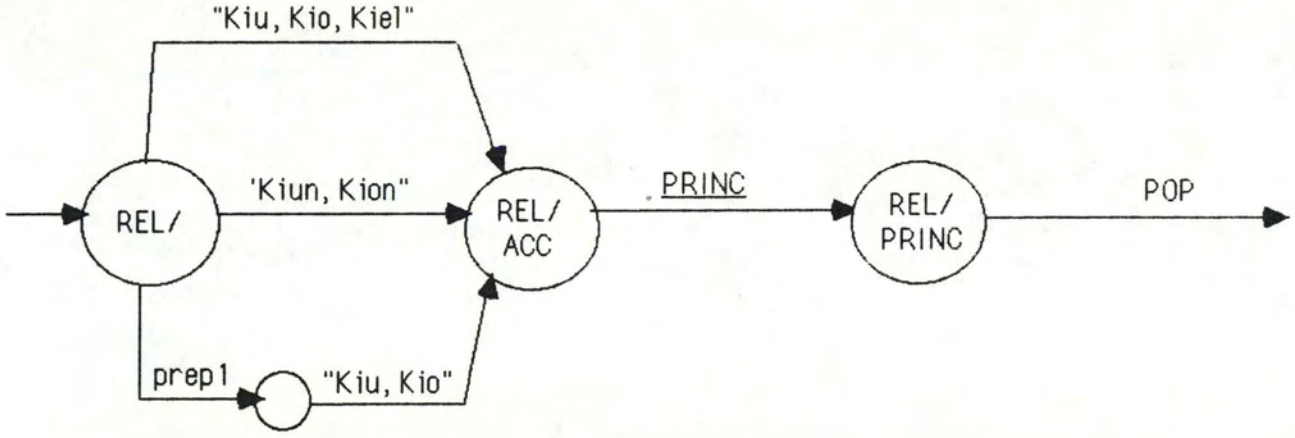


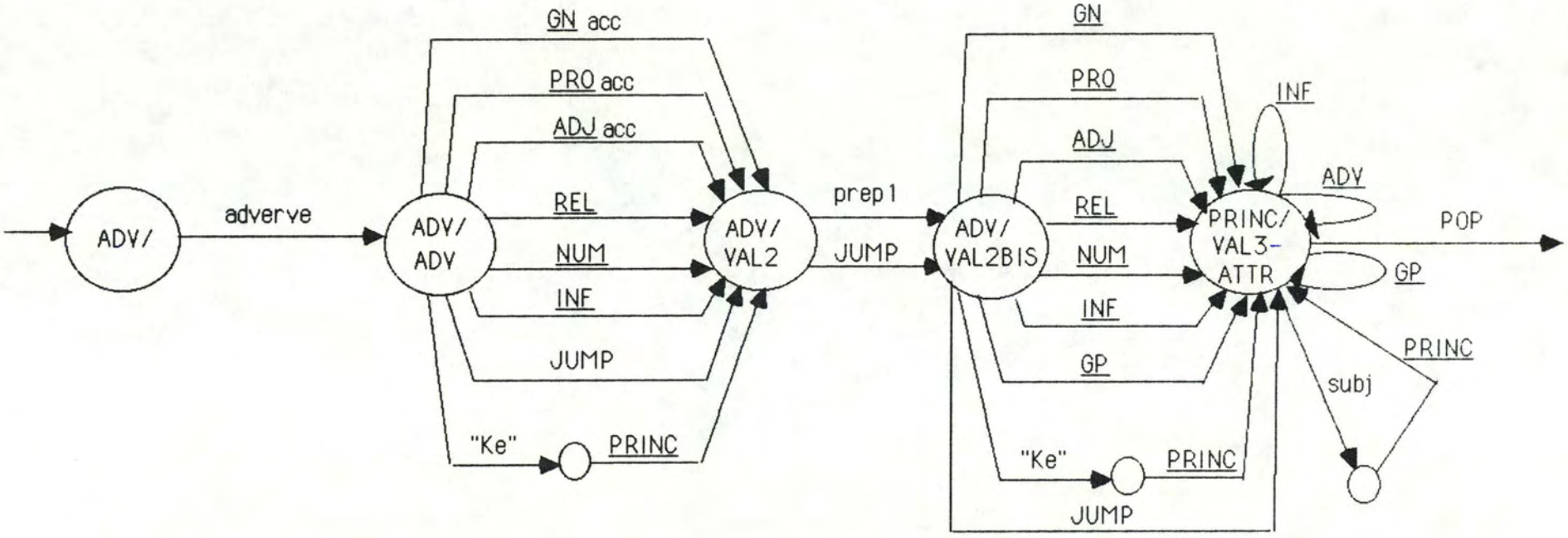
prep1 = (Al, De, Pri, Je)

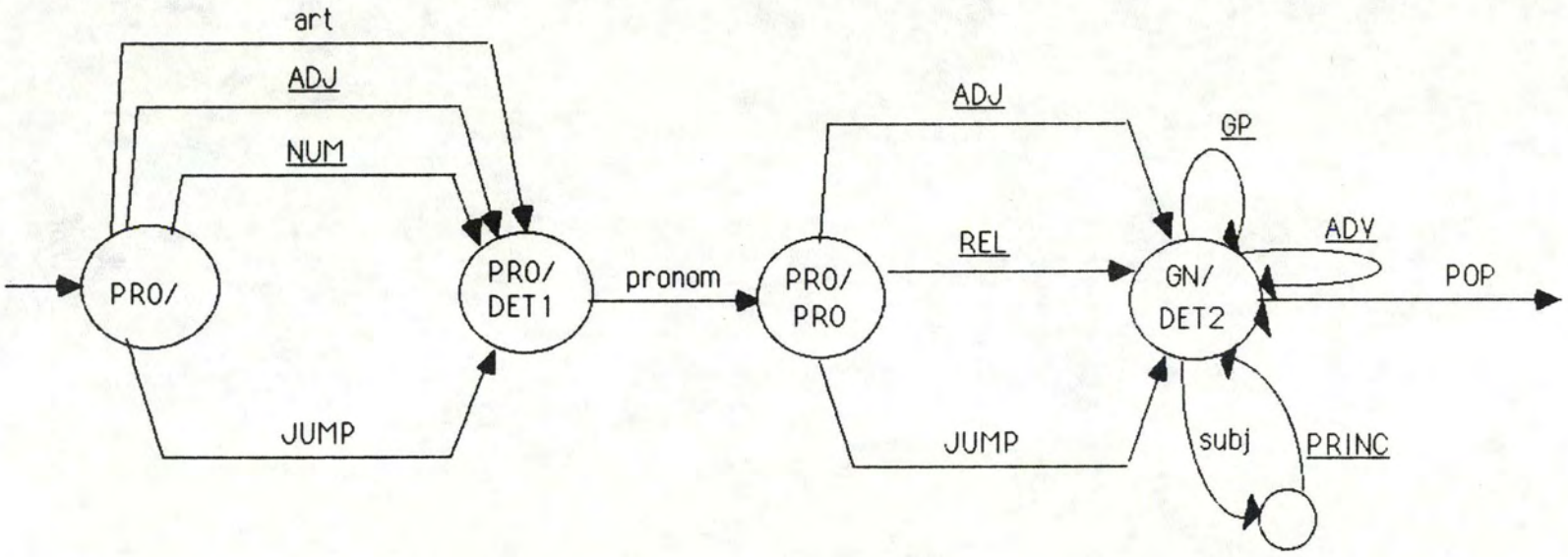


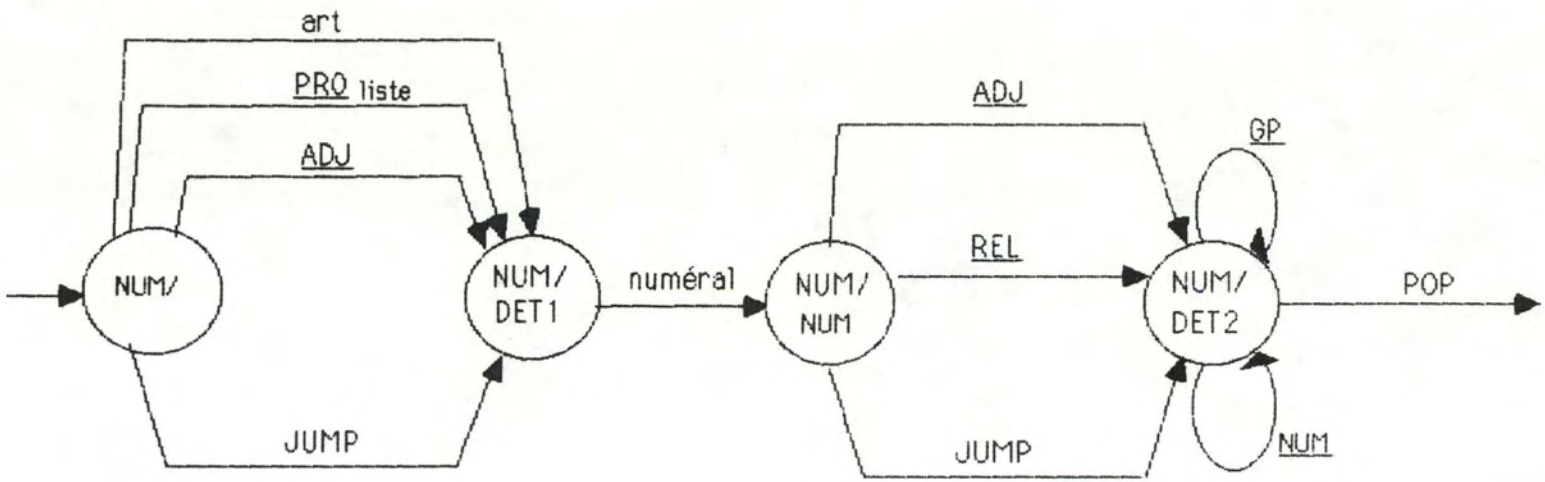
liste = (lo, lu, Tio, Tiu, Nenio, Neniu, Chio, Chiu)


INF/
-I
INF/
V
GN acc
PRO acc
ADJ acc
REL
NUM
INF
JUMP
"Ke"
PRINC
INF/
VAL2
prep 1
JUMP
INF/
VAL2BIS
GN
PRO
ADJ
REL
NUM
INF
GP Por
"Ke"
PRINC
JUMP
INF/
VAL3-
ATTR
INF
ADV
GP
subj
PRINC
POP


"Kiu, Kio, Kiel"
REL/
'Kiun, Kion"
REL/
ACC
PRINC
REL/
PRINC
POP
prep1
"Kiu, Kio"


ADV/
adverve
ADV/
ADV
GN acc
PRO acc
ADJ acc
REL
NUM
INF
JUMP
"Ke"
PRINC
ADV/
VAL2
prep1
JUMP
ADV/
VAL2BIS
GN
PRO
ADJ
REL
NUM
INF
GP
"Ke"
PRINC
JUMP
PRINC/
VAL3-
ATTR
INF
ADV
POP
GP
PRINC
subj


art
ADJ
NUM
PRO/
PRO/
DET1
JUMP
pronom
ADJ
REL
PRO/
PRO
JUMP
GP
ADV
GN/
DET2
POP
subj
PRINC



liste = (lo, lu, Tio, Tiu, Nenio, Neniu, Chio, Chiu)

Annexe 3 :

Le fichier des ATN

```
(PRINC/
  (PUSH GN/ T
    (SETR VAL1 *)
    (TO PRINC/VAL1))
  (PUSH PRO/ T
    (SETR VAL1 *)
    (TO PRINC/VAL1))
  (PUSH ADJ/ T
    (SETR VAL1 *)
    (TO PRINC/VAL1))
  (PUSH NUM/ T
    (SETR VAL1 *)
    (TO PRINC/VAL1))
  (PUSH REL/ T
    (SETR VAL1 *)
    (TO PRINC/VAL1))
  (WRD Ke T
    (SETR VAL1 (BUILDQ((SUBJ *))))
    (TO PRINC/KE1))
  (PUSH INF/ T
    (SETR VAL1 *)
    (TO PRINC/VAL1))
  (JUMP PRINC/VAL1 T
    (SETR VAL1 NIL)))

(PRINC/KE1
  (PUSH PRINC/ T
    (ADDR VAL1 *)
    (TO PRINC/VAL1)))

(PRINC/VAL1
  (CAT verbe T
    (SETR V LEX)
    (TO PRINC/V))
  (JUMP PRINC/V T
    (SETR V NIL)))

(PRINC/V
  (PUSH GN/ T
    (SETR VAL2 *)
    (TO PRINC/VAL2))
  (PUSH PRO/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
```

```
    (SETR VAL2 *)
    (TO PRINC/VAL2))
  (PUSH ADJ/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO PRINC/VAL2))
  (PUSH REL/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO PRINC/VAL2))
  (PUSH NUM/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO PRINC/VAL2))
  (PUSH INF/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO PRINC/VAL2))
  (WRD Ke (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 (BUILDQ((SUBJ *))))
    (TO PRINC/KE2))
  (JUMP PRINC/VAL2 T
    (SETR VAL2 NIL)))

(PRINC/KE2
  (PUSH PRINC/ T
    (ADDR VAL2 *)
    (TO PRINC/VAL2)))

(PRINC/VAL2
  (MEM (Al,De,Pri,Je) T
    (SETR VAL3 (BUILDQ((PREP *))))
    (SETR ALFLAG T)
    (TO PRINC/VAL2BIS))
  (JUMP PRINC/VAL2BIS (NULLR ALFLAG)
    (SETR VAL3 NIL)))

(PRINC/VAL2BIS
  (PUSH GN/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO PRINC/VAL3-AT))
```

```
  (PUSH PRO/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO PRINC/VAL3-AT))
  (PUSH ADJ/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO PRINC/VAL3-AT))
  (PUSH REL/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO PRINC/VAL3-AT))
  (PUSH NUM/ T
    (ADDR VAL3 *)
    (TO PRINC/VAL3-AT))
  (PUSH INF/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO PRINC/VAL3-AT))
  (PUSH GP/ (AND(EQUAL(*)(QUOTE Por))(NULLR ALFLAG))
    (SETR ATTR *)
    (TO PRINC/VAL3-AT))
  (WRD Ke T
    (COND ((NULLR VAL3)(SETR ATTR (BUILDQ((SUBJ *)))))
          (T (SETR VAL3 (BUILDQ((SUBJ *))))))
    (TO PRINC/VAL2-KE))
  (JUMP PRINC/VAL3-AT T
    (SETR ATTR NIL)
    (SETR VAL3 NIL)))

(PRINC/VAL2-KE
  (PUSH PRINC/ T
    (COND ((NULLR VAL3)(ADDR ATTR *))
          (T (ADDR VAL3 *)))
    (TO PRINC/VAL3-AT)))

(PRINC/VAL3-AT
  (PUSH INF/ (NULLR PASSAGE)
    (SETR CINF *)
    (SETR PASSAGE T)
    (TO PRINC/VAL3-AT))
  (PUSH ADV/ ADV-START
    (ADDR CIRC *)
    (TO PRINC/VAL3-AT))
  (PUSH GP/ PP-START
```

```
    (ADDR CIRC *)
    (TO PRINC/VAL3-AT))
  (CAT subjectif T
    (ADDR CIRC (BUILDQ((SUBJ *))))
    (TO PRINC/SUBJ))
  (POP (BUILDQ( PRINC (VAL1 +)(V +)(VAL2 +)(ATTR +)(VAL3 +)(COINF +)(CIRC +) )
VAL1 V VAL2 ATTR VAL3 CINF CIRC) T))

(PRINC/SUBJ
  (PUSH PRINC/ T
    (ADDR CIRC *)
    (TO PRINC/VAL3-AT)))

(GN/
  (CAT article T
    (SETR DET1 (BUILDQ((ART *))))
    (TO GN/DET1))
  (JUMP GN/DET1 T
    (SETR DET1 NIL))
  (PUSH ADJ/ T
    (SETR DET1 *)
    (TO GN/DET1))
  (PUSH NUM/ NUM-START
    (SETR DET1 *)
    (TO GN/DET1))
  (PUSH PRO/ PRO-START
    (VERIFY (APPARTIENT(GETR PRO)(lo,lu,Tio,Tiu,Nenio,Neniu,Chio,Chiu)))
    (SETR DET1 *)
    (TO GN/DET1)))

(GN/DET1
  (CAT substantif T
    (SETR N LEX)
    (TO GN/NOM)))

(GN/NOM
  (PUSH ADJ/ ADJ-START
    (SETR DET2 *)
    (TO GN/DET2))
  (PUSH REL/ T
    (SETR DET2 *)
    (TO GN/DET2))
  (JUMP GN/DET2 T
```

```
    (SETR DET2 NIL)))

(GN/DET2
  (PUSH INF/ T
    (ADDR COMP *)
    (TO GN/DET2))
  (CAT subjectif T
    (ADDR COMP (BUILDQ((SUBJ *))))
    (TO GN/DET2BIS))
  (JUMP GN/COMP T))

(GN/DET2BIS
  (PUSH PRINC/ T)
    (ADDR COMP *)
    (TO GN/COMP)))

(GN/COMP
  (PUSH GN/ SUB-START
    (ADDR CIRC *)
    (TO GN/COMP))
  (PUSH ADV/ T
    (ADDR CIRC *)
    (TO GN/COMP))
  (PUSH GP/ T
    (ADDR CIRC *)
    (TO GN/COMP))
  (POP (BUILDQ( GN (DET1 +)(N +)(DET2 +)(COMP +)(CIRC +) ) DET1 N DET2 COMP
CIRC) T))

(GP/
  (CAT preposition T
    (SETR PREP LEX)
    (TO GP/PREP)))

(GP/PREP
  (PUSH GN/ T
    (SETR CPREP *)
    (TO GP/COMP))
  (PUSH PRO/ PRO-START
    (SETR CPREP *)
    (TO GP/COMP))
  (PUSH INF/ T
    (SETR CPREP *)
```

```
    (TO GP/COMP))
  (PUSH NUM/ T
    (SETR CPREP *)
    (TO GP/COMP))
  (PUSH GP/ T
    (SETR CPREP *)
    (TO GP/COMP))
  (WRD Ke T
    (SETR CPREP (BUILDQ((SUBJ *))))
    (TO GP/PREP2))
  (WRD Chu  T
    (SETR CPREP (BUILDQ((PREP *))))
    (TO GP/PREP2))
  (JUMP GP/COMP T
    (SETR CPREP NIL)))

(GP/PREP2
  (PUSH PRINC/ T
    (ADDR CPREP *)
    (TO GP/COMP)))

(GP/COMP
  (PUSH ADV/ T
    (SETR CADV *)
    (TO GP/COMP))
  (POP (BUILDQ( GPREP (PREP +)(COPREP +)(CIADV +) ) PREP CPREP CADV) T))

(INF/
  (CAT verbe (CHECKF INFINITIF)
    (SETR V LEX)
    (TO INF/V)))

(INF/V
  (PUSH GN/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 *)
    (TO INF/VAL2))
   (PUSH PRO/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO INF/VAL2))
  (PUSH ADJ/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
```

```
    (SETR VAL2 *)
    (TO INF/VAL2))
  (PUSH REL/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO INF/VAL2))
  (PUSH NUM/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO INF/VAL2))
  (PUSH INF/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO INF/VAL2))
  (WRD Ke (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 (BUILDQ((SUBJ *))))
    (TO INF/KE2))
  (JUMP INF/VAL2 T
    (SETR VAL2 NIL)))

(INF/KE2
  (PUSH PRINC/ T
    (ADDR VAL2 *)
    (TO INF/VAL2)))

(INF/VAL2
  (MEM (Al,De,Pri,Je) (CATCHECK(GETR V)(QUOTE VALENCE 3))
    (SETR VAL3 (BUILDQ((PREP *))))
    (SETR ALFLAG T)
    (TO INF/VAL2BIS))
  (JUMP INF/VAL2BIS (NULLR ALFLAG)
    (SETR VAL3 NIL)))

(INF/VAL2BIS
  (PUSH GN/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO INF/VAL3-AT))
  (PUSH PRO/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO INF/VAL3-AT))
```

```
  (PUSH ADJ/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO INF/VAL3-AT))
  (PUSH REL/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO INF VAL3-AT))
  (PUSH NUM/ T
    (ADDR VAL3 *)
    (TO INF VAL3-AT))
  (PUSH INF/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO INF VAL3-AT))
  (PUSH GP/ (AND(EQUAL(*)(QUOTE Por))(NULLR ALFLAG))
    (SETR ATTR *)
    (TO INF/VAL3-AT))
  (WRD Ke T
    (COND ((NULLR VAL3)(SETR ATTR (BUILDQ((SUBJ *)))))
          (T (SETR VAL3 (BUILDQ((SUBJ *))))))
    (TO INF/VAL2-KE))
  (JUMP INF/VAL3-AT T
    (SETR ATTR NIL)
    (SETR VAL3 NIL)))

(INF/VAL2-KE
  (PUSH PRINC/ T
    (COND ((NULLR VAL3)(ADDR ATTR *))
          (T (ADDR VAL3 *)))
    (TO INF/VAL3-AT)))

(INF/VAL3-AT
  (PUSH INF/ (NULLR PASSAGE)
    (SETR CINF *)
    (SETR PASSAGE T)
    (TO INF/VAL3-AT))
  (PUSH ADV/ ADV-START
    (ADDR CIRC *)
    (TO INF/VAL3-AT))
  (PUSH GP/ PP-START
    (ADDR CIRC *)
    (TO INF/VAL3-AT))
  (CAT subjectif T
    (ADDR CIRC (BUILDQ((SUBJ *))))
```

```
    (TO INF/SUBJ))
  (POP (BUILDQ( INF (V +)(VAL2 +)(ATTR +)(VAL3 +)(COINF +)(CIRC +) ) V VAL2
ATTR VAL3 CINF CIRC) T))

(INF/SUBJ
  (PUSH PRINC/ T
    (ADDR CIRC *)
    (TO INF/VAL3-AT)))

(NUM/
  (CAT article T
    (SETR DET1 (BUILDQ((ART *))))
    (TO NUM/DET1))
  (PUSH PRO/ PRO-START
    (VERIFY (APPARTIENT(GETR PRO)(lo,lu,Tio,Tiu,Nenio,Neniu,Chio,Chiu)))
    (SETR DET1 *)
    (TO NUM/DET1))
  (PUSH ADJ/ ADJ-START
    (SETR DET1 *)
    (TO NUM/DET1))
  (JUMP NUM/DET1 T
    (SETR DET1 NIL)))

(NUM/DET1
  (CAT numeral T
    (SETR NUM LEX)
    (TO NUM/NUM)))

(NUM/NUM
  (PUSH ADJ/ ADJ-START
    (SETR DET2 *)
    (TO NUM/DET2))
  (PUSH REL/ T
    (SETR DET2 *)
    (TO NUM/DET2))
  (JUMP NUM/DET2 T
    (SETR DET2 *)))

(NUM/DET2
  (PUSH GP/ T
    (SETR CIRC *)
    (TO NUM/DET2))
  (PUSH NUM/ NUM-START
```

```
    (SETR CIRC *)
    (TO NUM/DET2))
  (POP (BUILDQ( GNUM (DET1 +)(NUM +)(DET2 +)(CIRC +) ) DET1 NUM DET2 CIRC) T))

(ADV/
  (CAT adverbe T
    (SETR ADV LEX)
    (TO ADV/ADV)))

(ADV/ADV
  (PUSH GN/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 *)
    (TO ADV/VAL2))
   (PUSH PRO/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADV/VAL2))
  (PUSH ADJ/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADV/VAL2))
  (PUSH REL/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADV/VAL2))
  (PUSH NUM/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADV/VAL2))
  (PUSH INF/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADV/VAL2))
  (WRD Ke (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 (BUILDQ((SUBJ *))))
    (TO ADV/KE2))
  (JUMP ADV/VAL2 T
    (SETR VAL2 NIL)))

(ADV/KE2
  (PUSH PRINC/ T
```

```
    (ADDR VAL2 *)
    (TO ADV/VAL2)))

(ADV/VAL2
  (MEM (A1,De,Pri,Je) (CATCHECK(GETR V)(QUOTE VALENCE 3))
    (SETR VAL3 (BUILDQ((PREP *))))
    (SETR ALFLAG T)
    (TO ADV/VAL2BIS))
  (JUMP ADV/VAL2BIS (NULLR ALFLAG)
    (SETR VAL3 NIL)))

(ADV/VAL2BIS
  (PUSH GN/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO ADV/VAL3-AT))
  (PUSH PRO/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO ADV/VAL3-AT))
  (PUSH ADJ/ T
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO ADV/VAL3-AT))
  (PUSH REL/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO ADV VAL3-AT))
  (PUSH NUM/ T
    (ADDR VAL3 *)
    (TO ADV VAL3-AT))
  (PUSH INF/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO ADV VAL3-AT))
  (PUSH GP/ (AND(EQUAL(*)(QUOTE Por))(NULLR ALFLAG))
    (SETR ATTR *)
    (TO ADV/VAL3-AT))
  (WRD Ke T
    (COND ((NULLR VAL3)(SETR ATTR (BUILDQ((SUBJ *)))))
          (T (SETR VAL3 (BUILDQ((SUBJ *))))))
    (TO ADV/VAL2-KE))
  (JUMP ADV/VAL3-AT T
    (SETR ATTR NIL)
    (SETR VAL3 NIL)))
```

```
(ADV/VAL2-KE
  (PUSH PRINC/ T
    (COND ((NULLR VAL3)(ADDR ATTR *))
       (T (ADDR VAL3 *)))
    (TO ADV/VAL3-AT)))

(ADV/VAL3-AT
  (PUSH INF/ (NULLR PASSAGE)
    (SETR CADV *)
    (SETR PASSAGE T)
    (TO ADV/VAL3-AT))
  (PUSH ADV/ ADV-START
    (ADDR CIRC *)
    (TO ADV/VAL3-AT))
  (PUSH GP/ PP-START
    (ADDR CIRC *)
    (TO ADV/VAL3-AT))
  (CAT subjectif T
    (ADDR CIRC (BUILDQ((SUBJ *))))
    (TO ADV/SUBJ))
  (POP (BUILDQ( GADV (ADV +)(VAL2 +)(ATTR +)(VAL3 +)(COINF +)(CIRC +) ) ADV
VAL2 ATTR VAL3 CINF CIRC) T))

(ADV/SUBJ
  (PUSH PRINC/ T
    (ADDR CIRC *)
    (TO ADV/VAL3-AT)))

(PRO/
  (CAT article T
    (SETR DET1 (BUILDQ((ART *))))
    (TO PRO/DET1))
  (PUSH ADJ/ ADJ-START
    (SETR DET1 *)
    (TO PRO/DET1))
  (PUSH NUM/ NUM-START
    (SETR DET1 *)
    (TO PRO/DET1))
  (JUMP PRO/DET1 T
    (SETR DET1 NIL)))

(PRO/DET1
```

```
  (CAT pronom T
    (SETR PRO LEX)
    (TO PRO/PRO)))

(PRO/PRO
  (PUSH ADJ/ T
    (SETR DET2 *)
    (TO PRO/DET2))
  (PUSH REL/ T
    (SETR DET2 *)
    (TO PRO/DET2 *))
  (JUMP PRO/DET2 T
    (SETR DET2 NIL)))

(PRO/DET2
  (PUSH GN/ T
    (ADDR CIRC *)
    (TO PRO/DET2))
  (PUSH ADV/ T
    (ADDR CIRC *)
    (TO PRO/DET2))
  (PUSH GP/ T
    (ADDR CIRC *)
    (TO PRO/DET2))
  (CAT subjectif T
    (ADDR (BUILDQ((SUBJ *))))
    (TO PRO/DET2-SUB))
  (POP (BUILDQ( GPRO (DET1 +)(PRO +)(DET2 +)(CIRC +) ) DET1 PRO DET2 CIRC) T))

(PRO/DET2-SUB
  (PUSH PRINC/ T
    (ADDR CIRC *)
    (TO PRO/DET2)))

(ADJ/
  (CAT article T
    (SETR DET1 (BUILDQ((ART *))))
    (TO ADJ/DET1))
  (PUSH NUM/ NUM-START
    (SETR DET1 *)
    (TO ADJ/DET1))
  (PUSH PRO/ PRO-START
    (SETR DET1 *)
```

```
    (TO ADJ/DET1))
  (JUMP ADJ/DET1 T
    (SETR DET1 NIL)))

(ADJ/DET1
  (PUSH ADV/ T
    (COND ((NULLR CADV)(SETR CADV *))
          (T (ADDR CADV *)))
    (TO ADJ/DET1 T))
  (JUMP ADJ/ADV T))

(ADJ/ADV
  (CAT adjectif T
    (COND ((NULLR ADJ)(SETR ADJ LEX))
          (T (ADDL ADJ LEX)))
    (TO ADJ/ADV))
  (PUSH GN/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
  (PUSH PRO/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
  (PUSH REL/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
  (PUSH NUM/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
  (PUSH INF/ (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR
V)(QUOTE VALENCE3)))
    (SETR VAL2 *)
    (TO ADJ/VAL2))
  (WRD Ke (OR(CATCHECK(GETR V)(QUOTE VALENCE2))(CATCHECK(GETR V)(QUOTE
VALENCE3)))
    (SETR VAL2 (BUILDQ((SUBJ *))))
    (TO ADJ/KE2))
  (JUMP ADJ/VAL2 T
    (SETR VAL2 NIL)))
```

```
(ADJ/KE2
  (PUSH PRINC/ T
    (ADDR VAL2 *)
    (TO ADJ/VAL2)))

(ADJ/VAL2
  (MEM (Al,De,Pri,Je) (CATCHECK(GETR V)(QUOTE VALENCE 3))
    (SETR VAL3 (BUILDQ((PREP *))))
    (SETR ALFLAG T)
    (TO ADJ/VAL2BIS))
  (JUMP ADJ/VAL2BIS (NULLR ALFLAG)
    (SETR VAL3 NIL)))

(ADJ/VAL2BIS
  (PUSH GN/ SUB-START
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO ADJ/VAL3-AT))
  (PUSH PRO/ PRO-START
    (COND ((NULLR VAL3)(SETR ATTR *))
          (T (ADDR VAL3 *)))
    (TO ADJ/VAL3-AT))
  (PUSH REL/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO ADJ VAL3-AT))
  (PUSH NUM/ NUM-START
    (ADDR VAL3 *)
    (TO ADJ VAL3-AT))
  (PUSH INF/ (EQUAL(GETR ALFLAG)(QUOTE T))
    (ADDR VAL3 *)
    (TO ADJ VAL3-AT))
  (PUSH GP/ (AND(EQUAL(*)(QUOTE Por))(NULLR ALFLAG))
    (SETR ATTR *)
    (TO ADJ/VAL3-AT))
  (WRD Ke T
    (COND ((NULLR VAL3)(SETR ATTR (BUILDQ((SUBJ *)))))
          (T (SETR VAL3 (BUILDQ((SUBJ *))))))
    (TO ADJ/VAL2-KE))
  (JUMP ADJ/VAL3-AT T
    (SETR ATTR NIL)
    (SETR VAL3 NIL)))

(ADJ/VAL2-KE
```

```
  (PUSH PRINC/ T
    (COND ((NULLR VAL3)(ADDR ATTR *))
       (T (ADDR VAL3 *)))
    (TO ADJ/VAL3-AT)))

(ADJ/VAL3-AT
  (PUSH INF/ (NULLR PASSAGE)
    (SETR CINF *)
    (SETR PASSAGE T)
    (TO ADJ/VAL3-AT))
  (PUSH ADV/ ADV-START
    (ADDR CIRC *)
    (TO ADJ/VAL3-AT))
  (PUSH GP/ PP-START
    (ADDR CIRC *)
    (TO ADJ/VAL3-AT))
  (CAT subjectif T
    (ADDR CIRC (BUILDQ((SUBJ *))))
    (TO ADJ/SUBJ))
  (POP (BUILDQ( GADJ (DET1 +)(COADV +)(ADJ +)(COINF +)(CIRC +) ) DET1 CADV
ADJ COINF CIRC) T))

(ADJ/SUBJ
  (PUSH PRINC/ T
    (ADDR CIRC *)
    (TO ADJ/VAL3-AT)))


(REL/
  (MEM (Kio,Kiu,Kies) T
    (SETR COR (BUILDQ((CORR *))))
    (TO REL/COR))
  (MEM (Kiun,Kion) T
    (SETR COR (BUILDQ((CORR *))))
    (TO REL/COR))
  (MEM (Al,De,Pri,Je) T
    (SETR PRED (BUILDQ((PREP *))))
    (TO REL/CAL)))

(REL/CAL
  (MEM (Kio,Kiu) T
    (ADDR PRINC (BUILDQ((CORR *))))
    (TO REL/COR)))
```

```
(REL/COR
  (PUSH PRINC/ T
    (ADDR PRINC *)
    (TO REL/PRINC)))

(REL/PRINC
  (POP (BUILDQ( REL (PREP +)(COR +)(PRINC +) ) PREP COR PRINC) T))
```

Annexe 4 :

Les programmes de traduction anglais-Esperanto

```
#include "[inf.mem.djamme._modules]mylib.c"
#include "[inf.mem.djamme._modules]chaines.c"
#include "[inf.mem.djamme._modules]elemrep.c"
#include "[inf.mem.djamme._modules]repertoire.c"
#include "[inf.mem.djamme._modules]dialogue.c"
#include "[inf.mem.djamme._montraducteur]primstrings.c"
#include "[inf.mem.djamme._montraducteur]selectphra.c"
#include "[inf.mem.djamme._montraducteur]tradphra.c"
#include "[inf.mem.djamme._montraducteur]affiche.c"
#include "[inf.mem.djamme._montraducteur]affichecontex.c"
#include "[inf.mem.djamme._montraducteur]traduire.c"

#define TAILLEPHRASE 160


/*************************************************************
*                                                            *
*        PROGRAMME de traduction des phrases qui apparaissent *
*        au moins 10 fois dans le repertoire                 *
*                                                            *
*************************************************************/

main()

{
 InitCdC();
 InitDialogue();
 InitElemRep();


 OuvrirFichiers();
 OuvrirDico();

 selectphrases();    /* selection de ces phrases */
 traduitphrases();   /* traduction de ces phrases */

 FermerFichiers();
 FermerDico();
 ClôturerElemRep();
 ClôturerDialogue();
 ClôturerCdC();

}
```

```
/*****************************************************************
*            OUVERTURE DES FICHIERS                              *
*****************************************************************/


OuvrirFichiers()

{
 Str2PoserQuestion("Donner le nom du fichier repertoire : ",NomRepIn);
 if ((RepIn = fopen(NomRepIn,"r")) == NULL)
    StopErreur("Le fichier repertoire ne veut pas s'ouvrir...");
 if ((Foccur = fopen("occur.dat","w")) == NULL)
    StopErreur("Le fichier des phrases selectionnees ne veut pas s'ouvrir...");
 if ((Ftrad = fopen("occurtrad.dat","w")) == NULL)
    StopErreur("Le fichier des traductions ne veut pas s'ouvrir...");
 return(0);
}


OuvrirDico()
{
 Str2PoserQuestion("Nom du fichier dictionnaire: ",NomDict);
 if ((Dict=fopen(NomDict,"r")) == NULL)
   StopErreur("Le fichier dictionnaire ne veut pas s'ouvrir...");
}




/*****************************************************************
*            FERMETURE DES FICHIERS                              *
*****************************************************************/

FermerFichiers()
{
 fclose(RepIn);
 fclose(Foccur);
 fclose(Ftrad);
```

```
}

FermerDico()
{
 fclose(Dict);
}


/*************************************************************
*                                                            *
*        PROGRAMME de sélection des phrases qui apparaissent *
*        au moins 10 fois dans le repertoire                 *
*                                                            *
*************************************************************/

FILE *Foccur;

struct node
{
    char *Phrase;
    int occurence;
    struct node *phragauche;
    struct node *phradroite;
};


selectphrases()

{ int a;
  struct node *racine, *arbre();
  CdC Phraln;
  char Phra[TAILLEPHRASE];

  InitChaine(&Phraln);
  racine = NULL;
  while ((a=lireelt(&Phraln))==0)
  {
   transf_en_char(&Phraln,Phra);
   racine = arbre(racine, Phra);
   AbandonnerChaine(&Phraln);
   InitCdC();
   InitChaine(&Phraln);
```

```
  }
  AbandonnerChaine(&PhraIn);
  imprimerarbre(racine);
  free(racine);
}




struct node *arbre(r, p)

struct node *r;
char *p;

{ int c;
  char *rp;

  if (r == NULL)
  {
   r = calloc(1,sizeof(struct node));
   rp = malloc(strlen(p)+1);
   strcpy(rp,p);
   r->Phrase = rp;
   r->occurence = 1;
   r->phragauche = r->phradroite = NULL;
  }
  else  if ((c = strcmp(p,r->Phrase)) == 0)
            r->occurence++;
  else  if ((c = strcmp(p,r->Phrase)) < 0)
            r->phragauche = arbre(r->phragauche,p);
        else
            r->phradroite = arbre(r->phradroite,p);

  return(r);
}




lireelt(ptPhraseIn)
```

```
CdC *ptPhraseIn;

{ int a;
  PhraseTrad PhraseTra;
  ContextePhrase ContexPhrase;
  InitPhraseTrad(&PhraseTra);
  InitContextePhrase(&ContexPhrase);
  EtatRep=Ouvert;
  if ((a=PrendreElt(&PhraseTra,&ContexPhrase))==0)
  {
    GetPhrase(&PhraseTra,ptPhraseIn);
    AbandonnerPhraseTrad(&PhraseTra);
    AbandonnerContextePhrase(&ContexPhrase);
    return(0);
  }
  else
    AbandonnerPhraseTrad(&PhraseTra);
    AbandonnerContextePhrase(&ContexPhrase);
    return(1);
}



imprimerarbre(r)

struct node *r;

{ if (r != NULL)
  {
    imprimerarbre(r->phragauche);
    if (r->occurence >= 10)
    {
       fputs(r->Phrase, Foccur);
       putc('\n',Foccur);
    }
    imprimerarbre(r->phradroite);
  }
}



transf_en_char(ptPhrasein,ptresult)
```

```
CdC *ptPhrasein;
char *ptresult;

{ int i;
  char c;
  for (i=0; (i <= (LongChaine(ptPhrasein))) ;i++)
  {    c = GetCarChaine(ptPhrasein,i);
       *ptresult++ = c;
  }
}
```

```
/*****************************************************************
*                                                                *
*          PROGRAMME QUI TRADUIT LES PHRASES QUI SONT PRESENTES  *
*                                                                *
*              AU MOINS 10 FOIS DANS LE REPERTOIRE               *
*****************************************************************/


FILE *Ftrad;


traduitphrases()

/* Procédure qui code la traduction des phrases qui sont presentes au moins
   10 fois dans le repertoire, et reecrit ces phrases et leur traduction dans
   le fichier occurtrad.                    */


{
    PhraseTrad pt;
    ContextePhrase cxt;
    CdC Phra, Trad;
    char c,
    phrase[TAILLEPHRASE],
    traduc[TAILLEPHRASE];
    int i;

    rewind(Foccur);  /* repositionne le fichier Foccur au debut */
    while (1)        /* boucle infinie (on en sort par break ... */
    {
      for (i=0; ((c=getc(Foccur)) != '\n') && (c != EOF); i++)
           phrase[i]=c;
      phrase[i]='\0';
      if (c == EOF) break;
     InitPhraseTrad(&pt);
     InitContextePhrase(&cxt);
      InitChaine(&Phra);
      InitChaine(&Trad);
      AffecterChaine(phrase, &Phra);
      PutPhrase(&Phra,&pt);
      AfficherPhrase(&pt);
      Traduire(&pt, &cxt);
      GetTrad1(&pt, &Trad);
```

```
        transf_en_char(&Trad, traduc);
        fputs(phrase, Ftrad);
        putc('\n', Ftrad);
        putc('#', Ftrad);
        fputs(traduc, Ftrad);
        putc('\n', Ftrad);
        AbandonnerChaine(&Phra);
        AbandonnerChaine(&Trad);
        AbandonnerPhraseTrad(&pt);
        AbandonnerContextePhrase(&cxt);
      }
}

/*-------------------------------------------------------------*/
```

```
Traduire(ptPhraTra,ptContexPhra)

PhraseTrad *ptPhraTra;
ContextePhrase *ptContexPhra;

{ CdC Trad;

  while ((StrReponseAlternative("\n\nVoulez vous un mot du dictionnaire? ")) ==
OUI)
  {
   ChercheMotsDico();
   AfficherPhrase(ptPhraTra);
  }
   AfficherPhrase(ptPhraTra);
   AfficherContexte(ptPhraTra,ptContexPhra);
   InitChaine(&Trad);
   StrPoserQuestion("\n\nIntroduisez la traduction de la phrase: ",&Trad);
   PutTrad1(&Trad,ptPhraTra);
   AbandonnerChaine(&Trad);
}
```

```
/*****************************************************************/
/*                                                               */
/*          PROCEDURE QUI AFFICHE A L'ECRAN LA PHRASE A TRADUIRE */
/*                                                               */
/*****************************************************************/

AfficherPhrase(PhraTra)

PhraseTrad *PhraTra;

{
 CdC Phra;

 InitChaine(&Phra);
 GetPhrase(PhraTra,&Phra);
 StrAfficherInfo("\nLa phrase :   ");
 AfficherInfo(&Phra);
 AbandonnerChaine(&Phra);
 printf("\n");
 return(0);
}

/*---------------------------------------------------------------*/
```

```
/*****************************************************************/
/*                                                               */
/*        EXTRAIRE D'UN DICTIONNAIRE UN MOT (DONNE EN ENTREE)    */
/*        ET SES TRADUCTIONS                                     */
/*                                                               */
/*****************************************************************/


#define REUSSI TRUE
#define RATE FALSE


/***************************FICHIERS*****************************/

FILE    *Dict;
        /* Pointeur du fichier dictionnaire */

   char NomDict[40];
        /* Nom du fichier dictionnaire */



/***************** PROGRAMME PRINCIPAL ************************/

ChercheMotsDico()
{
        bool TemoinDict;
        char Mot[80], MotDict[80], DescrMotDict[80];
        int rescomp, a;
        bool TemoinLire, TrouveMot;

        TrouveMot=FALSE;
        rewind(Dict); /* repositionne le fichier au debut */
        Str2PoserQuestion("Introduisez le mot : ", Mot);
        while ((TemoinLire = LireMotDict(MotDict,DescrMotDict)) == REUSSI)
        {
          rescomp=strcmp(Mot,MotDict);
         if(rescomp<0)
         {
            if (TrouveMot == FALSE)
                StrAlerter("Le mot ne se trouve pas dans le dictionnaire!!!");
```

```
            break;
          }
          else if(rescomp == 0)
          {
              AfficherMot(MotDict,DescrMotDict);
               TrouveMot=TRUE;
          }
        }
        printf("\n");
}


/********** LIRE UN MOT DU DICTIONNAIRE ET SA DESCRIPTION *********/


LireMotDict(MotDict,DescrMotDict)

char MotDict[],DescrMotDict[];
{
    char c;
    int i;
    bool DansMot;

    if((c=getc(Dict))==EOF)
    return(RATE);
    else
    {
        DansMot=TRUE;
        i=0;

         do {
              if(isalpha(c))
              {

    if(DansMot==TRUE) MotDict[i++]=c;

    else DescrMotDict[i++]=c;
              }
              else
              {

    if(DansMot==TRUE)
```

```
{
        DansMot=FALSE;
 MotDict[i]='\0';
 i=0;
}
if(c=='\n')
{
 DescrMotDict[i]='\0';
 return(REUSSI);
}
else
{
 DescrMotDict[i++]=c;
}
            }
      }     while((c=getc(Dict))!= EOF);

      ungetc(c,Dict);
      if(DansMot==TRUE)
      {
            if(i==0)
            {
return(RATE);
            }
            else
            {
MotDict[i]='\0';
```

```
    DescrMotDict[0]='\0';

    return(REUSSI);
                }
          }
          else
          {
                DescrMotDict[i]='\0';
                return(REUSSI);
          }
     }
}


/************** SORTIR UN MOT ET SON DESCRIPTEUR *****************/


AfficherMot(MotDict,DescrMotDict)

char MotDict[],DescrMotDict[];

/* On sort le mot et son descripteur sur une même ligne.    */
{
printf("\n%s",MotDict);
printf("\t%s\n",DescrMotDict);
}
```

```
/*****************************************************************/
/*                                                               */
/*          PROCEDURE QUI AFFICHE A L'ECRAN LE CONTEXTE DE LA    */
/*                                                               */
/*              PHRASE A TRADUIRE                                */
/*                                                               */
/*****************************************************************/


AfficherContexte(PhraTra,ContexPhra)

PhraseTrad *PhraTra;
ContextePhrase *ContexPhra;

{
 PhraseTrad PhCont;
 CdC Phra2, Trad;
 int i;

 StrAfficherInfo("\nSon contexte : ");
 for (i=0;i < LgContexte(ContexPhra);i++)
 {
   InitPhraseTrad(&PhCont);
   GetEltContexte(ContexPhra,i,&PhCont);
   InitChaine(&Phra2);
   GetPhrase(&PhCont,&Phra2);
   AfficherInfo(&Phra2);
   StrAfficherInfo("\n          ");
   AbandonnerChaine(&Phra2);
   AbandonnerPhraseTrad(&PhCont);
 }
 InitChaine(&Trad);
 GetTrad1(PhraTra,&Trad);
 if (LongChaine(&Trad) != 0)
 {
  StrAfficherInfo("\nSa traduction: ");
  AfficherInfo(&Trad);
 }
 AbandonnerChaine(&Trad);
 return(0);
}

/*---------------------------------------------------------------*/
```

```
/*****************************************************************
*                                                                *
*  PROGRAMME qui ecrit dans le repertoire les phrases deja traduites     *
*  (c.a.d les phrases du fichier occurtrad.dat), et qui traduit et ecrit dans  *
*  le repertoire les phrases concernant l'heure (11 a.m, 6 h.,...)       *
*                                                                *
*****************************************************************/

FILE *Ftrad;


main()

{
 InitCdC();
 InitDialogue();
 InitElemRep();

 EtatRep=Ferme;
 OuvrirRepertoire();
 InitPreSession();
 PreSession();
 FermerRepertoire();
 ClôturerElemRep();
 ClôturerDialogue();
 ClôturerCdC();

}
```

```
/*****************************************************************
*
*  INITIALISE LA PRE-SESSION DE TRADUCTION: ouvre occurtrad.dat en lecture
*
*****************************************************************
```

```
/

InitPreSession()

{
 if ((Ftrad = fopen("occurtrad.dat","r")) == NULL)
    StopErreur("Le fichier des phrases selectionnees ne veut pas s'ouvrir...");

}


/*****************************************************************/

PreSession()

{
    PhraseTrad PhraTra;
    ContextePhrase ContexPhra;
    CdC Phra, Trad;
    char phrase[TAILLEPHRASE],
         PhraRep[TAILLEPHRASE],
         traduc[TAILLEPHRASE];
    int result, a, p;

    Etat0 :    InitPhraseTrad(&PhraTra);
               InitContextePhrase(&ContexPhra);
               goto Etat1;

    Etat1 :    result=PrendreElt(&PhraTra,&ContexPhra);
               goto Etat2;



    Etat2 :
         if (result != 0) goto Etat5;
         else
         {
                InitChaine(&Trad);
                InitChaine(&Phra);
                GetPhrase(&PhraTra, &Phra);
                transf_en_char(&Phra, PhraRep);
                if (isdigit(PhraRep[0]))
                {
```

```
                if ((TraduitHeure(PhraRep, traduc)) == 0)
                   goto Etat3;
                else
                   goto Etat4;
              }
              else
              {
               rewind(Ftrad);
               while ((p=PrendrePhrase(phrase)) == 0)
               {
                if ((a=strcmp(phrase,PhraRep)) == 0)
                {
                   PrendreTraductionFich(traduc);
                   goto Etat3;
                }
               }
               goto Etat4;
              }
        }

    Etat3 :
              AffecterChaine(traduc, &Trad);
              PutTrad1(&Trad, &PhraTra);
              goto Etat4;


   Etat4 :    RendreElt(&PhraTra);
              AbandonnerChaine(&Trad);
              AbandonnerChaine(&Phra);
              AbandonnerPhraseTrad(&PhraTra);
              AbandonnerContextePhrase(&ContexPhra);
              goto Etat0;


   Etat5 :    return(0);
}

/*-----------------------------------------------------------------*/
```

```
PrendrePhrase(p)

char *p;

{ char c;
  int i;

   if (c == '#')
     for (i=0; (c=getc(Ftrad)) != '\n' ;i++);

   if ((c=getc(Ftrad)) == EOF) return(-1);

   i=0;
   do {
       p[i]=c;
       i++;
     } while ((c=getc(Ftrad)) != '\n' );

   p[i]='\0';
   return(0);
}



PrendreTraductionFich(t)

char *t;

{ int i=0;
  char c;

  if ((c=getc(Ftrad)) != '#')
    StopErreur("Erreur dans le fichier des traductions: traduction sans -#-!!!");
  while ((c=getc(Ftrad)) != '\n')
  {
      t[i]=c;
      i++;
  }
  t[i]='\0';

}
```

```
TraduitHeure(p,c)

char *p, *c;

{ int i, j, k;
  char mot[TAILLEPHRASE];

  for (i=0; isdigit(p[i]) || p[i] == ' ';i++)   c[i]=p[i];

  for (j=i,k=0; p[j] != '\0' ;j++,k++)
     mot[k]=p[j];
  mot[k]='\0';
  if ((strcmp(mot,"a.m.")) != 0 && (strcmp(mot,"p.m.")) != 0 && (strcmp(mot,"h."))
!= 0)
     return(-1);
  else
  {
    for (i; p[i] != '\0';i++)
    {
      if (p[i] == 'a' || p[i] == 'p')
      {
        c[i]=p[i];
        i++;
        c[i]='t';
      }
      else
         c[i]=p[i];
    }
    c[i]='\0';
    return(0);
  }
}
```

```
/*****************************************************************
*                                                                *
*  PROGRAMME QUI REALISE LA TRADUCTION PROPREMENT-DITE           *
*                                                                *
*****************************************************************/


main()

 InitCdC();
 InitDialogue();
 InitElemRep();


 EtatRep=Ferme;
 OuvrirRepertoire();
 OuvrirDico();
 Session();
 StrAfficherPasseTemps("Veuillez patienter SVP...");
 FermerRepertoire();
 FermerDico();
 SupprimerPasseTemps();
 CloturerElemRep();
 CloturerDialogue();
 CloturerCdC();

}

/*****************************************************************/
```

```
/* PROCEDURE qui code l'automate d'un session de traduction */

Session()

{
    PhraseTrad PhraTra;
    ContextePhrase ContexPhra;
    CdC Phra, Trad;
    char phrase[TAILLEPHRASE],
    traduc[TAILLEPHRASE];
    int result, a, p;

    Etat0 :    InitPhraseTrad(&PhraTra);
               InitContextePhrase(&ContexPhra);
               goto Etat1;

    Etat1 :    result=PrendreElt(&PhraTra,&ContexPhra);
               goto Etat2;

    Etat2 :
          if (result != 0) goto Etat6;
          else if (!PoursuivreSession()) goto Etat5;
            else
            {
               InitChaine(&Trad);
               InitChaine(&Phra);
               GetPhrase(&PhraTra, &Phra);
               transf_en_char(&Phra, phrase);
               AfficherPhrase(&PhraTra);
               AfficherContexte(&PhraTra,&ContexPhra);
               GetTrad1(&PhraTra,&Trad);
               printf("\n");
               if ((LongChaine(&Trad)) == 0 ||
(StrReponseAlternative("Voulez-vous modifier cette traduction ?")) == OUI)
                   goto Etat3;
               else
                   goto Etat4;
          }

      Etat3 :
          Traduire(&PhraTra,&ContexPhra);
            goto Etat4;
```

```
    Etat4 :
            RendreElt(&PhraTra);
            AbandonnerChaine(&Trad);
            AbandonnerChaine(&Phra);
            AbandonnerPhraseTrad(&PhraTra);
            AbandonnerContextePhrase(&ContexPhra);
            goto Etat0;

    Etat5 :
            RendreElt(&PhraTra);
            AbandonnerChaine(&Trad);
            AbandonnerChaine(&Phra);
            AbandonnerPhraseTrad(&PhraTra);
            AbandonnerContextePhrase(&ContexPhra);
            goto Etat6;


    Etat6 :
         return(0);
}


/****** FONCTIONS D'INITIALISATION ET DE CLOTURE DES FICHIERS *******/

OuvrirDico()

{
    Str2PoserQuestion("Nom du fichier dictionnaire : ",NomDict);

    if((Dict=fopen(NomDict,"r"))==NULL)
         StopErreur("Le fichier dictionnaire ne veut pas s'ouvrir...\n");
}


FermerDico()

/* Procédure qui clôture le fichier dictionnaire */

{
    fclose(Dict);
}
```

```
/*-----------------------------------------------------------------*/

PoursuivreSession()

/* Fonction qui renvoie TRUE si l'utilisateur veut poursuivre la session,
  FALSE sinon.
     */

{
 if ((StrReponseAlternative("\nOn continue la session de traduction ? ")) ==
OUI)
    return(TRUE);
 else
     return(FALSE);
}
```

Annexe 5 :

Les fonctions-C de l'analyseur syntaxique

```
FILE    *ATN,           /* pointeur du fichier des ATN  */
        *Dict;          /* Pointeur du fichier dictionnaire   */


char NomDict[40];       /* Nom du fichier dictionnaire */

char tab[13][4] = { {"At"},{"It"},{"Ot"},{"Ant"},{"Int"},
    {"Ont"},{"J"},{"N"},{"As"},{"Is"},{"Os"},{"U"},{"Us"} };

    /* tableau des suffixes esperanto (conjugaisons et pluriel) */


char pref[18][6] = { {"Dis"},{"Ek"},{"For"},{"Mis"},{"Re"}, {"Retro"},
    {"Vir"},{"Bo"},{"Eks"},{"Ge"},{"Mal"}, {"Pra"},
    {"Fi"},{"Ne"},{"Vic"},{"Chef"},{"Pse□do"}, {"Fush"} };

    /* tableau des prefixes esperanto */


typedef char etat[20];

typedef char typemot[30];


struct registre {

        char nomreg[20];
        char valreg[250];
        struct registre *regnext;
            };


struct transition {

        char nomarc[5];
        typemot mot;
        char test[160];
        char action[160][6];
        char forme[160];
        char liste[20][15];
        etat etatsuivant;
        struct transition *nextarc;
            };
```

```
struct stack {

        struct registre *regsup;
        struct transition *arcstack;
        struct stack *nextstack;
             };


struct configuration {

        int string;
        int indicfin;
        struct registre *acregs;
        struct stack *acstack;
        int lexmode;
        etat acetat;
        struct configuration *acnext;
                };


struct alternative {

        struct configuration *altconfig;
        struct transition *altarc;
        char altlex[500];
        char altraclex[30];
        struct alternative *altnext;
             };


struct parses {

        char structure[500];
        struct parses *nextparse;
             };


struct configuration *accour,*acdern,*acnewdern;

struct alternative *altcour;
```

```
struct registre *regprem,*regcour,*regdern;

struct parses *premparse,*dernparse,*courparse;

struct stack *premstack;

char lex[500], raclex[30];

char phrase[240];

int finphrase=0,pointeurphrase=0;
```

```
#include "[inf.mem.djamme._parser._saveparser]arcs.c"
#include "[inf.mem.djamme._parser._saveparser]conditions.c"
#include "[inf.mem.djamme._parser._saveparser]actions.c"
#include "[inf.mem.djamme._parser._saveparser]lexic.c"
#include "[inf.mem.djamme._parser._saveparser]step.c"


Parser(phrase)

char *phrase;

{
 struct configuration *acfs;
 struct alternative *alt;
 int l;

 if (altcour == NULL)
 {
  acfs=calloc(1,sizeof(struct configuration));
  acfs->acstack=NULL;
  acfs->acregs=NULL;
  strcpy(acfs->acetat,"PRINC/");
  acfs->acnext=NULL;
  acfs->lexmode=1;
  acnewdern=acfs;
  goto etat0;
 }
 else
  goto etat3;

etat0: if (acfs->lexmode == 1)
     {
      if ((l=Lexic(phrase,lex,raclex)) == 1)
      {
       printf("\nLE MOT %s N'EST PAS DANS LE DICO !!! \n",lex);
       return(1);
      }
      else
      {
       if (l==2)
        goto etat2;
      }
     }
     goto etat1;

etat1:
```

```
        Step(acfs,altcour,1);
        if (acfs != acnewdern)
        {
         acfs=acnewdern;
         goto etat0;
        }
        else
        {
         goto etat3;
        }

etat2:
        Step(acfs,altcour,1);
        if (acfs != acnewdern)
        {
         acfs=acnewdern;
         goto etat2;
        }
        if (premparse == NULL)
        {
         goto etat3;
        }
        else
        {
         return(0);
        }

etat3:
        alt=altcour;
        altcour=altcour->altnext;
        Step(acfs,alt,2);
        if (acfs != acnewdern)
        {
         acfs=acnewdern;
         if (finphrase == 0)
          goto etat0;
         else
          if (premparse == NULL)
           goto etat2;
        }

        if (premparse == NULL)
        {
          if (altcour == NULL)
          {
           return(1);
```

```
        }
        else
        {
          cfree(alt);
          goto etat3;
        }
      }
      else
      {
        return(0);
      }
}
```

```
int pos;

Lexic(phrase,mot,racine)

char *phrase, *mot, *racine;

{ int i,j,L;
  char c;

  if (phrase[pointeurphrase] == '\0')
  {
    finphrase=1;
    strcpy(mot,"\0");
    strcpy(racine,"\0");
    return(2);
  }
  for (i=pointeurphrase,j=0; (c=phrase[i]) != ' ' && c != ',' && c != '.' &&
          c != ';' && c != ':' && c != '\0' ;i++,j++)

      mot[j]=phrase[i];
  mot[j]='\0';
  for (i++; phrase[i] == ' ' ;i++);   /* sauter les blancs eventuels */

  pointeurphrase=i;
  return(Morph(mot,racine));
}



Morph(mot,racine)

char *mot, *racine;

{
  int i, j, k, l, m, p, Estpref, rescomp, okmot, okpart ;
  char nouvmot[30], prefdico[6], prefmot[30], suffmot[6],suffdico[6];
  char motdico[30], descrmot[800], ligne[80], suffsuiv[6], suivant[30],
maj[7];
  char suff[6][6];

  strcpy(nouvmot,mot);
  for (i=0,k=0; mot[i] != '\0' ;i++)
```

```
{
  if (isupper(mot[i]))
  {
    maj[k]=i;
    k++;
  }
}
maj[k]=i;
maj[k+1]='\0';
Estpref=0;

if (ChercheMotDict(mot) == 0)
{
 strcpy(racine,mot);
 return(0);
}

for (i=0; i != maj[1] ;i++)
 prefmot[i] = mot[i];
prefmot[i] = '\0';
for (l=0; l != 18 ;l++)
{
 if (strcmp(prefmot,pref[l]) == 0)
 {
   for (i=maj[1],j=0; i != maj[2] ;i++,j++)
    suivant[j]=mot[i];
   suivant[j]='\0';

   if (ChercheMotDict(suivant) == 0)
   {
    for(i=maj[1],j=0; i != k; i++,j++)
     nouvmot[j]=mot[i];
    nouvmot[j]='\0';
    strcpy(racine,suivant);
    Estpref=1;
   }
 }
}
if (Estpref == 0)
{
 strcpy(racine,prefmot);
 p=1;
```

```
}
else
 p=2;

rewind(Dict);
while (LireMotDict(motdico,descrmot) == 0)
{
   if ((rescomp=strcmp(motdico,racine)) > 0)
    return(1);
   else
    if (rescomp == 0)
     break;
}

i=maj[p];
j=0;
l=0;
while (p != k)
{

 if (i == maj[p+1])
 {
   suff[j][l] ='\0';
   j=0;
   l++;
   p++;
 }
 else
 {
   suff[j][l]=mot[i];
   j++;
   i++;
 }
}
suff[0][l]='\0';
pos=0;
while (LireLigneMot(descrmot,ligne) == 0)
{
 i=0;
 if (Estpref == 1)
 {
  for (i=1,j=0;ligne[i] != '*';i++,j++)
```

```
  prefdico[j]=ligne[i];
 prefdico[j]='\0';
 if (strcmp(prefdico,prefmot) != 0)
  continue; /* ligne suivante */
}
for (i;ligne[i] != 'x';i++);
i=i+1;
l=0;
while (suff[0][l] != '\0')
{
 okmot=0;
 for(i++,j=0;ligne[i]!='*' && ligne[i]!=' ' && ligne[i]!='\0'; i++,j++)
  suffdico[j]=ligne[i];
 suffdico[j]='\0';
 for (k=0,l;suff[k][l] != '\0';k++)
  suffmot[k]=suff[k][l];
 suffmot[k]='\0';
 l++;
 if (strcmp(suffdico,suffmot) == 0)
  continue;

 if (strcmp(suffdico,"I") == 0 )
 {
  for (m=0; m != 6;m++)
  {
   if (strcmp(suffmot,tab[m]) == 0)
   {
    for (k=0,l;suff[k][l] != '\0';k++)
     suffmot[k]=suff[k][l];
    suffmot[k]='\0';
    if (strcmp(suffmot,"A") == 0 || strcmp(suffmot,"E") == 0)
      return(0);
    else
      okmot=1;
   }
  }
  if (okmot == 0)
  {
   for (m=8;m!=13;m++)
   {
    if (strcmp(suffmot,tab[m]) == 0)
      return(0);
```

```
        }
       }
      }
      if (strlen(suffdico) == 0 )
      {
       if (strcmp(suffmot,"J") == 0 )
        continue;
       else
       {
        if (strcmp(suffmot,"N") == 0)
         return(0);
       }
      }
      okmot=1;
    }
    if (okmot == 0) return(0);
  }
  return(1);
}



ChercheMotDict(mot)

char *mot;

{ int rescomp,
     TemoinLire;
  char motdico[30],descrmot[160];

          rewind(Dict);
          while ((TemoinLire = LireMotSansDescr(motdico)) == 0)
          {
            rescomp=strcmp(mot,motdico);
            if (rescomp < 0)
               return(1);
            else
              if(rescomp == 0)
                return(0);
          }
}
```

```
LireMotSansDescr(MotDict)

char *MotDict;
{
     char c;
     int i, DansMot;

     if((c=getc(Dict))==EOF)   return(1);
     else
     {
          DansMot=0;
          i=0;

          do {
               if(isalpha(c) )
               {
                    if(DansMot==0) MotDict[i++]=c;
               }
               else
               {
                    if(DansMot==0)
                    {
                         DansMot=1;
                         MotDict[i]='\0';
                         i=0;
                    }
                    if(c=='&')
                    {
                         return(0);
                    }
               }
          } while((c=getc(Dict))!= EOF);

          ungetc(c,Dict);
          if(DansMot==0)
          {
               if(i==0)
               {
                    return(1);
               }
```

```
                else
                {
                    MotDict[i]='\0';
                    return(0);
                }
            }
        }
}


LireMotDict(MotDict,DescrMotDict)

char *MotDict, *DescrMotDict;
{
    char c;
    int i, DansMot;

    if((c=getc(Dict))==EOF)  return(1);
    else
    {
        DansMot=0;
        i=0;

        do {
            if(isalpha(c) )
            {
                if(DansMot==0) MotDict[i++]=c;
                else DescrMotDict[i++]=c;
            }
            else
            {
                if(DansMot==0)
                {
                    DansMot=1;
                    MotDict[i]='\0';
                    i=0;
                }
                if(c=='&')
                {
                    DescrMotDict[i]='\0';
                    return(0);
                }
```

```
                    else
                    {
                        DescrMotDict[i++]=c;
                    }
                }
            } while((c=getc(Dict))!= EOF);

            ungetc(c,Dict);
            if(DansMot==0)
            {
                if(i==0)
                    return(1);
                else
                {
                    MotDict[i]='\0';
                    DescrMotDict[0]='\0';
                    return(0);
                }
            }
            else
            {
                DescrMotDict[i]='\0';
                return(0);
            }
        }
}


LireLigneMot(descrmot,ligne)

char *descrmot, *ligne;
{
 int i;
 if (descrmot[pos] == '\0')
  return(1);
 for (i=0,pos;descrmot[pos] != '=';i++,pos++)
  ligne[i]=descrmot[pos];
 ligne[i]='\0';
 for (pos;descrmot[pos] != '\n';pos++);
 pos++;
 return(0);
}
```

```
Step(conf,alt,type)

struct configuration *conf;
struct alternative *alt;
int type;

{
 struct configuration *ac, *acint;
 struct alternative *newalt, *altdern, *altprem;
 struct transition *arc;
 struct stack *pushstack;
 struct parses *parsepop;
 int i,j;
 char motcourant[160], zone[1000];

 if (type == 2)
 {
  ac=alt->altconfig;
  arc=alt->altarc;
  strcpy(lex,alt->altlex);
  strcpy(raclex,alt->altraclex);
  finphrase=ac->indicfin;
  pointeurphrase=ac->string;
 }
 else
 {
  ac=conf;
  arc=ChercheArcs(ac->acetat,arc);
 }

 regcour=ac->acregs;
 while (arc != NULL)
 {
     if (strcmp(arc->nomarc,"JUMP") != 0 && strcmp(arc->nomarc,"POP") !=
0 && finphrase == 1)
       break;

      if (Conditions(arc->test) == 0)
      {
       if (strcmp(arc->nomarc,"CAT") == 0)
       {
        if (Cat(arc->mot,lex,raclex) != 0)
        {
```

```
      break;
    }
  }
  else
  {
   if (strcmp(arc->nomarc,"MEM") == 0)
   {
    if (Mem(arc->liste) != 0)
    {
     break;
    }
   }
   else
   {
    if (strcmp(arc->nomarc,"WRD") == 0)
    {
     if (strcmp(lex,arc->mot) != 0)
     {
      break;
     }
    }
   }
  }
  acint=calloc(1,sizeof(struct configuration));
  if (strcmp(arc->nomarc,"PUSH") == 0)
  {
   pushstack=calloc(1,sizeof(struct stack));
   pushstack->regsup=ac->acregs;
   pushstack->arcstack=arc;
   if (ac->acstack == NULL)
      pushstack->nextstack=NULL;
   else
      pushstack->nextstack=ac->acstack;
   acint->acstack=pushstack;
   strcpy(acint->acetat,arc->mot);
  }
  else
  {
   acint->acstack=ac->acstack;
   strcpy(motcourant,lex);
   if (strcmp(arc->nomarc,"POP") == 0)
   {
    EvalueForme(arc->forme,zone);
```

```
        if (finphrase == 1 && ac->acstack == NULL)
        {
         parsepop=calloc(1,sizeof(struct parses));
         strcpy(parsepop->structure,zone);
         parsepop->nextparse=NULL;
         if (premparse == NULL)
          premparse=parsepop;
         else
          dernparse->nextparse=parsepop;
         dernparse=parsepop;
        }
        else
        {
         strcpy(lex,zone);
         arc=ac->acstack->arcstack;
         regdern=regcour;
         regcour=ac->acstack->regsup;
         if (ac->acstack->nextstack == NULL)
          acint->acstack=NULL;
         else
          acint->acstack=ac->acstack->nextstack;

        }
       }

       if (ExecuteAction(arc) != 0)
       {
        strcpy(lex,motcourant);
        free(acint);
        break;
       }
       strcpy(acint->acetat,arc->etatsuivant);
       strcpy(lex,motcourant);
      }

      acint->acnext=NULL;
      if (strcmp(arc->nomarc,"JUMP") == 0 || strcmp(arc->nomarc,"POP") ==
0 || strcmp(arc->nomarc,"PUSH") == 0 )
          acint->lexmode=0;
      else
          acint->lexmode=1;

      acint->acregs=regcour;
```

```
        acnewdern->acnext=acint;
        acnewdern=acint;
      }
      break;
   }

  if (type == 1)
  {
       altdern=NULL;
       arc=arc->nextarc;
       while (arc != NULL)
       {
        newalt=calloc(1,sizeof(struct alternative));
        newalt->altconfig=ac;
        newalt->altconfig->string=pointeurphrase;
        newalt->altconfig->indicfin=finphrase;
        newalt->altarc=arc;
        strcpy(newalt->altlex,lex);
        strcpy(newalt->altraclex,raclex);
        if (altdern == NULL)
         altprem=newalt;
        else
         altdern->altnext=newalt;
        altdern=newalt;
        arc=arc->nextarc;
       }
       if (altdern != NULL)
       {
        altdern->altnext=altcour;
        altcour=altprem;
       }
   }

}


Mem(liste)

char liste[20][15];

{
 int i,j;
 char element[20];
```

```
 for (i=0,j=0;liste[0][j] != '\0';)
 {
  element[i]=liste[i][j];
  if (liste[i][j] == '\0')
  {
   if (strcmp(lex,element) == 0)
    return(0);
   else
   {
    i=0;
    j++;
   }
  }
  else
    i++;
 }
 return(1);
}



Cat(mot,testmot,racinemot)

char *mot, *testmot, *racinemot;

{ int i,j,k,l,Estpref,okverbe, rescomp;
  char suffmot[6], motdico[30], categ[15], descrmot[800], cat[5], ligne[80];
  char suff[6][6];

  if (strcmp(testmot,racinemot) == 0)
  {
   rewind(Dict);
   while(LireMotDict(motdico,descrmot)== 0)
   {
    if ((rescomp=strcmp(motdico,racinemot)) == 0)
    {
     pos=0;
     while(LireLigneMot(descrmot,ligne) == 0)
     {
      for(i=0;ligne[i] != '[';i++);
      for(i++,j=0;ligne[i] != ' ' && ligne[i] != '=' && ligne[i] != '*';i++,j++)
       cat[j]=ligne[i];
```

```
cat[j]='\0';
if (ligne[i] == '*')
    break;
if (strcmp(cat,"Ce") == 0)
 strcpy(categ,"subjectif");
else
{
 if (strcmp(cat,"Ae") == 0)
  strcpy(categ,"article");
 else
 {
  if (strcmp(cat,"Ai") == 0)
   strcpy(categ,"adjectif");
  else
  {
   if (strcmp(cat,"P") == 0)
    strcpy(categ,"preposition");
   else
   {
    if (strcmp(cat,"Aa") == 0)
     strcpy(categ,"adverbe");
    else
    {
     if (strcmp(cat,"Ne") == 0)
      strcpy(categ,"pronom");
     else
      if (strcmp(cat,"Nu") == 0)
       strcpy(categ,"numeral");
      else
       if (strcmp(cat,"B") == 0)
        strcpy(categ,"abreviation");
       else
        break;
    }
   }
  }
 }
 if (strcmp(categ,mot)==0)
   return(0);
} /* fin while (LireLigneMot(descrmot... */

} /* fin if (strcmp(motdico,racinemot)==0) */
```

```
    else
    {
      if (rescomp > 0)
       break;
    }

  } /* fin while (LireMotDict(motdico... */
  return(1);

} /* fin if (strcmp(testmot,racinemot)==0) */

i=0;
j=0;
while (racinemot[i] != '\0')
{
 if (racinemot[i] == testmot[j])
 {
  k=i;
  for(;racinemot[k] == testmot[j];k++,j++);
  if (racinemot[k] == '\0' && (isupper(testmot[j]) || testmot[j] == '\0'))
   break;
  else
   j--;
 }
 j++;
}
l=0;
while (testmot[j] != '\0')
{
 suff[0][l]=testmot[j];
 for(j++,i=1;!(isupper(testmot[j])) && testmot[j] != '\0';i++,j++)
   suff[i][l]=testmot[j];
 suff[i][l]='\0';
 l++;
 i=0;
}
suff[0][l]='\0';
l=0;
okverbe=1;
while (suff[0][l] != '\0')
{
 for(k=0,l;suff[k][l] != '\0';k++)
   suffmot[k]=suff[k][l];
```

```
    suffmot[k]='\0';
    if (strcmp(suffmot,"O") == 0 )
     strcpy(categ,"substantif");
    else
    if (strcmp(suffmot,"A") == 0 )
     strcpy(categ,"adjectif");
    else
    if (strcmp(suffmot,"E") == 0 )
     strcpy(categ,"adverbe");
    else
    if (strcmp(suffmot,"I") == 0 )
     strcpy(categ,"verbe");
    else
    {
     for (i=8;i != 13;i++)
     {
       if (strcmp(suffmot,tab[i]) == 0)
       {
        strcpy(categ,"verbe");
        okverbe=0;
       }
     }
     if (okverbe == 1)
     {
      l++;
      continue;
     }
    }
    if (strcmp(categ,mot)==0)
      return(0);
    else
      break;
  }
  return(1);

}
```

```
Conditions(test)

char *test;

{ char typecond[12], start[6];
  int i,j;

  for (i=0;test[i] == ' ' || test[i] == '(' ;i++);
  for (i,j=0;test[i] != ' ' && test[i] != '(' && test[i] != ')' && test[i] !=
'\0';i++,j++)
      typecond[j]=test[i];
  typecond[j]='\0';


  if (strcmp(typecond,"T") == 0) return(0);
  else
  {
   if (strcmp(typecond,"CATCHECK") == 0)
     return(1);


/*
    return(Cond_Catcheck(test));
*/


   else
   {
    if (strcmp(typecond,"NULLR") == 0)
     return(Cond_Nullr(test));
    else
    {
     if (strcmp(typecond,"OR") == 0)
      return(Cond_Or(test));
     else
     {
      if (strcmp(typecond,"EQUAL") == 0)
       return(Cond_Equal(test));
      else
      {
       if (strcmp(typecond,"AND") == 0)
        return(Cond_And(test));
```

```
        else
        {
         if (strcmp(typecond,"APPARTIENT") == 0)
          return(Cond_Appartient(test));
         else
          if (strcmp(typecond,"CHECKF") == 0)
           return(1);
        }
       }
      }
     }
    }
  }

/* verifie les conditions de type START */

  for (i=0;typecond[i] != '-' && typecond[i] != '\0' ;i++);
  if (typecond[i] == '-')
  {

/*
   for (i++,j=0; typecond[i] != '\0' ;i++,j++)
     start[j]=typecond[i];
   start[j]='\0';
   if (strcmp(start,"START") == 0)
*/

    return(Cond_Start(test));
  }
}


Cond_Or(test)

char *test;

{
  char cond[160];
  int i,j,npg,npd;

  npg=0;
  npd=0;
  for (i=0;test[i]==' ' || test[i]=='(';i++);
```

```
  for (i;test[i]!='(';i++);
  j=0;
  while (test[i]!='\0')
  {
   if (test[i]=='(') npg++;
   else
    if (test[i]==')') npd++;
   cond[j]=test[i];
   if (npg==npd)
   {
    cond[++j]='\0';
    if (Conditions(cond) == 0)
        return(0);
    else
        j=0;
   }
   else
    j++;

   i++;
  }
  return(1);
}
```

```
Cond_And(test)

char *test;

{
 char cond[160];
 int i,j,npg,npd;

 npg=0;
 npd=0;
 for (i=0;test[i]==' ' || test[i]=='(';i++);
 for (i;test[i]!='(';i++);
 j=0;
 while (test[i]!='\0')
```

```
  {
   if (test[i]=='(') npg++;
   else
    if (test[i]==')') npd++;
   cond[j]=test[i];
   if (npg==npd)
   {
    cond[++j]='\0';
    if (Conditions(cond) != 0)
        return(1);
    else
        j=0;
   }
   else
    j++;

   i++;
  }
  return(0);
}




Cond_Equal(test)

char *test;

{
 char forme1[100], forme2[100], eval1[200], eval2[200];
 int i,j,npg,npd;

 npg=0;
 npd=0;
 for (i=0;test[i]==' ' || test[i]=='(';i++);
 for (i;test[i]!=' ' && test[i]!='(';i++);
 for (i,j=0;test[i]!='\0' ;i++,j++)
 {
  if (test[i]=='(') npg++;
  else
   if (test[i]==')') npd++;
  forme1[j]=test[i];
```

```
    if (npg==npd)
      break;
  }
  forme1[++j]='\0';

  for (i++,j=0;test[i]!='\0' ;i++,j++)
  {
    if (test[i]=='(') npg++;
    else
      if (test[i]==')') npd++;
    forme2[j]=test[i];
    if (npg==npd)
      break;
  }
  forme2[++j]='\0';

  if (EvalueForme(forme1,eval1)==0 && EvalueForme(forme2,eval2)==0)
  {
    if (strcmp(eval1,eval2) == 0)
      return(0);
  }
  return(1);
}


EvalueForme(forme,eval)

char *forme, *eval;

{
  char  typeforme[6], reg[20], car[20], val[200];
  int i,j,k,l,npg,npd,npg2,npd2;

  i=0;
  if (forme[i] == '(' )
    i++;
  for (i,j=0;forme[i] != ' ' && forme[i] != ')' && forme[i] != '(';i++,j++)
  {
      typeforme[j]=forme[i];
  }
  typeforme[j]='\0';

  if (strcmp(typeforme,"NIL")== 0 || strcmp(typeforme,"T") == 0)
```

```
  strcpy(eval,typeforme);
 else
 {
  if (strcmp(typeforme,"*") == 0 || strcmp(typeforme,"LEX") == 0)
    strcpy(eval,lex);
  else
  {
   if (strcmp(typeforme,"QUOTE")== 0)
   {
    for (i;forme[i]==' ';i++);
    for (i,j=0;forme[i]!=')';i++,j++)
     eval[j]=forme[i];
    eval[j]='\0';
   }
   else
   {
    if (strcmp(typeforme,"GETR") == 0)
    {
     for (i;forme[i]==' ';i++);
     for (i,j=0;forme[i]!=' ' && forme[i]!=')';i++,j++)
       reg[j]=forme[i];
     reg[j]='\0';
     return(LireReg(reg,eval));
    }
    else
    {
     if (strcmp(typeforme,"BUILDQ") == 0)
     {
        for (i;forme[i] != '(';i++);
        npg=npd=0;
        npg2=1;
        npd2=0;
        for (j=i+1;npd2 != npg2 ;j++)
        {
          if (forme[j] == '(') npg2++;
          else
            if (forme[j] == ')') npd2++;
        }

        for (i,l=0;forme[i] != '\0';i++)
        {
         if(forme[i] == '+')
         {
```

```
            for (j;forme[j] == ' ';j++);
            for (j,k=0;forme[j] != ' ' && forme[j] != ')';k++,j++)
             reg[k]=forme[j];
            reg[k]='\0';

            if (LireReg(reg,val) == 0)
            {
             for (k=0,l;val[k] != '\0';k++,l++)
              eval[l]=val[k];
            }
           }
           else
           {
            if (forme[i] == '*')
            {
             for (k=0,l;lex[k] != '\0';k++,l++)
              eval[l]=lex[k];
            }
            else
            {
             eval[l]=forme[i];
             l++;
             if (forme[i] == '(') npg++;
             else
             if (forme[i] == ')') npd++;
             if (npg==npd)
              break;
            }
           }
          }
          eval[l]='\0';
         }
        }
       }
    }
   return(0);
}


LireReg(nom,val)
```

```
char *nom, *val;

{
 struct registre *regpt;

 regpt=regcour;
 while (regpt != NULL)
 {
  if (strcmp(regpt->nomreg,nom) == 0)
  {
     strcpy(val,regpt->valreg);
     return(0);
  }
  regpt=regpt->regnext;
 }
 return(1);
}



Cond_Nullr(test)

char *test;

{
 char val[200], reg[20];
 int i,j;

 for (i=0;test[i] != ' ' ;i++);
 for (i;test[i] == ' ' ;i++);
 for (i,j=0;test[i] != ' ' && test[i] != ')' ;i++,j++)
    reg[j]=test[i];
 reg[j]='\0';

 if ((LireReg(reg,val) == 1) || strcmp(val,"NIL") == 0 || val[0] == '\0')
 {
     return(0);
 }
 return(1);
}


Cond_Appartient(test)
```

```
char *test;

{
 char forme[100], element[30], eval[200];
 int i,j,npg,npd;

 npg=0;
 npd=0;


 for (i=0;test[i]==' ' || test[i]=='(';i++);
 for (i;test[i]!=' ' && test[i]!='(';i++);
 for (i,j=0;test[i]!='\0' ;i++,j++)
 {
  if (test[i]=='(') npg++;
  else
   if (test[i]==')') npd++;
  forme[j]=test[i];
  if (npg==npd)
    break;
 }
 forme[++j]='\0';

 if (EvalueForme(forme,eval) == 0)
 {

  for (++i,j=0;test[i]!=')' ;i++)
  {
   if (test[i]==',')
   {
    element[j]='\0';
    j=0;
    if (strcmp(element,eval) == 0)
       return(0);
   }
   else
   {
    if (test[i] !='(' && test[i] != ' ')
    {
      element[j]=test[i];
      j++;
    }
```

```
    }
   }
   element[j]='\0';
   if (strcmp(element,eval) == 0)
     return(0);
  }
  return(1);
}


Cond_Start(test)

char *test;

{
 char typestart[4], cat[15];
 int i,j;

 for (i=0,j=0;test[i] != '-' ;i++,j++)
    typestart[j]=test[i];
 typestart[j]='\0';
 if (strcmp(typestart,"PRO") == 0)
  strcpy(cat,"pronom");
 else
 {
  if (strcmp(typestart,"NUM") == 0)
   strcpy(cat,"numeral");
  else
  {
   if (strcmp(typestart,"ADV") == 0)
    strcpy(cat,"adverbe");
   else
   {
    if (strcmp(typestart,"PP") == 0)
     strcpy(cat,"preposition");
    else
    {
     if (strcmp(typestart,"ADJ") == 0)
      strcpy(cat,"adjectif");
     else
     {
      if (strcmp(typestart,"SUB") == 0)
       strcpy(cat,"substantif");
```

```
        }
       }
      }
     }
    }
    return(Cat(cat,lex,raclex));

}
```

```
struct transition *ChercheArcs(eta,arc)

etat eta;
struct transition *arc;

{
 char descretat[3000];
 etat etatlu;
 int j,i,l,cmp,premier;
 struct transition *arclu,*arcour;

 rewind(ATN);
 premier=0;
 while ((l=LireEtat(etatlu,descretat)) == 0)
 {
  if ((cmp=strcmp(etatlu,eta)) == 0)
  {
   arcour=NULL;
   arclu=calloc(1,sizeof(struct transition));
   while ((l=LireArc(descretat,arclu)) == 0)
   {
       if (premier==0)
       {
          arc=arclu;
          premier=1;
       }
       else
       {
          if (premier==1)
          {
           arc->nextarc=arclu;
           premier=2;
           arcour=arclu;
          }
          else
          {
           arcour->nextarc=arclu;
           arcour=arclu;
          }
       }
       arclu=calloc(1,sizeof(struct transition));
```

```
    }
    arcour->nextarc=NULL;
    return(arc);
   }

 }
 return(NULL);
}


LireEtat(etatlu,descretatlu)

etat etatlu;
char *descretatlu;

{
 char c;
 int i,Dansetat,npg,npd;

 while ((c=getc(ATN)) != '(')
 {
  if (c==EOF) return(1);
 }
 Dansetat=0;
 i=0;
 npg=npd=0;
 while ((c=getc(ATN)) != EOF)
 {
  if (Dansetat==0)
  {
   if (c != '\n')
   {
    etatlu[i]=c;
    i++;
   }
   else
   {
    Dansetat=1;
    etatlu[i]='\0';
    i=0;
   }
  }
```

```
    else
    {
     if (c == '(') npg++;
     else
      if (c == ')') npd++;
     if ((npg+1) == npd)
     {
      descretatlu[i]='\0';

      return(0);
     }
     descretatlu[i]=c;
     i++;
    }
   }
   return(1);
}


LireArc(descretat,arc)

char *descretat;
struct transition *arc;

{
 static int i=0;
 int j,k,npg,npd,npg2,npd2, Dansarc,Dansmot,
     Danspop,Dansmem,Danstest,Dansaction;
 char mot[30];

 for (i;descretat[i] != '(' && descretat[i] != '\0';i++);
 if (descretat[i] == '\0')
 {
     i=0;          /* remet i a 0 car c'est une variable statique... */
     return(1);
 }
 for (++i;descretat[i] == ' ';i++);
 Dansarc=Dansmot=Dansaction=0;
 Danspop=Dansmem=Danstest=1;
 j=k=0;
 npg=npd=npg2=npd2=0;
 while (descretat[i] != '\0')
```

```
{
 if (Dansarc == 0)
 {
  if (descretat[i] != ' ' && descretat[i] != '(')
  {
   arc->nomarc[j]=descretat[i];
   j++;
  }
  else
  {
   Dansarc=1;
   arc->nomarc[j]='\0';
   for (i;descretat[i] == ' ';i++);
   if (strcmp(arc->nomarc,"POP") == 0)
    Danspop=0;
   if (strcmp(arc->nomarc,"MEM") == 0)
    Dansmem=0;
   j=0;
   k=0;
   i--;
  }
 }
 else
 {
  if (Danspop==0)
  {
   for (i;descretat[i] != '(';i++);
   i++;
   j=0;
   npg++;
   npg2=1;
   while (npg2 != npd2)
   {
    if (descretat[i] == '(')
    {
     npg++;
     npg2++;
    }
    if (descretat[i] == ')')
    {
     npd++;
     npd2++;
```

```
    }
    arc->forme[j]=descretat[i];

    j++;
    i++;
  }
  arc->forme[j]='\0';
  Dansmot=Danspop=Dansaction=1;
  Danstest=0;
 }
 else
 {
  if (Dansmem == 0)
  {
   if (descretat[i]==',')
   {
    arc->liste[j][k]='\0';
    k++;
    j=0;
   }
   else
   {
    if (descretat[i] == ' ' || descretat[i] == '(');
    else
    {
     if (descretat[i] == ')')
     {
      arc->liste[j][k]='\0';
      arc->liste[0][++k]='\0';
      Dansmem=Dansmot=1;
      Danstest=0;
     }
     else
     {
      arc->liste[j][k]=descretat[i];
      j++;
     }
    }
   }
  }
  else
  {
```

```
      if (Dansmot==0)
      {
       if (descretat[i] == ' ')
       {
        Dansmot=1;
        Danstest=0;
        mot[j]='\0';
        if (strcmp(arc->nomarc,"CAT")==0 || strcmp(arc->nomarc,"WRD")==0 ||
strcmp(arc->nomarc,"PUSH")==0)
            strcpy(arc->mot,mot);
        if (strcmp(arc->nomarc,"JUMP")==0)
            strcpy(arc->etatsuivant,mot);
       }
       else
       {
        mot[j]=descretat[i];
        j++;

       }
      }
      else
      {
       if (Danstest==0)
       {
        for (i;descretat[i] == ' ';i++);
        for (i,j=0;descretat[i] != '\n' && descretat[i] != '\0' ;i++,j++)
         arc->test[j]=descretat[i];
        arc->test[j]='\0';
        i--;
        Danstest=1;
       }
       else
       {
        if (Dansaction == 0)
        {
           for (i;descretat[i] == ' ' || descretat[i] == '\n' || descretat[i] ==
'\0';i++);
           if (descretat[i] == '\0')
           {
            arc->action[0][0]='\0';
            return(0);
           }
```

```
            npg=npd=0;
            for (i,j=0,k=0;descretat[i] != '\0';i++)
            {
             if (descretat[i] == '(')
               npg++;
             else
             {
              if (descretat[i] == ')')
                npd++;
             }

             if ((npg+1) == npd)
             {
              arc->action[j][k]='\0';
              return(0);
             }
             arc->action[j][k]=descretat[i];
             if (npg==npd && npg != 0)
             {
              arc->action[++j][k]='\0';  /* pour indiquer la fin d'une action */
              npg=npd=0;
              k++;
              j=0;
              if (descretat[++i] != ')')
                  for(i;descretat[i] == ' ' || descretat[i] == '\n';i++);
              --i;
             }
             else
             {
              j++;
             }
            }
           }
          }
         }
        }
       }
      }
      i++;
     }
     return(0);
}
```

```
ExecuteAction(arc)

struct transition *arc;

{
 int i,j;
 char action[160];

 i=j=0;
 while (arc->action[0][j] != '\0')
 {
  if (arc->action[i][j] == '\0')
  {
   j++;
   action[i]='\0';
   if (ExecuteUneAction(action,arc) != 0)
     return(1);
   i=0;
  }
  action[i]=arc->action[i][j];
  i++;
 }

 return(0);
}



ExecuteUneAction(action,arc)

char *action;
struct transition *arc;

{
 int i,j;
 char typeaction[10];

 for (i=1,j=0;action[i] != ' ' && action[i] != '(' && action[i] != '\0';i++,j++)
    typeaction[j]=action[i];
 typeaction[j]='\0';
 if (strcmp(typeaction,"TO") == 0)
 {
  To(action,arc);
```

```
    return(0);
  }
  else
  {
   if (strcmp(typeaction,"SETR") == 0)
   {
    Setr(action);
    return(0);
   }
   else
   {
    if (strcmp(typeaction,"ADDR") == 0)
    {
     Addr(action);
     return(0);
    }
    else
    {
     if (strcmp(typeaction,"COND") == 0)
     {
      Cond(action,arc);
      return(0);
     }
     else
     {
      if (strcmp(typeaction,"VERIFY") == 0)
       return(Verify(action));
      else
      {
       if (strcmp(typeaction,"ADDL") == 0)
       {
          Addl(action);
          return(0);
       }
      }
     }
    }
   }
  }
}


Setr(action)
```

```
char *action;

{ int i,j,npg,npd;
  char nom[20],forme[100],eval[500];
  struct registre *reg;

  for (i=0;action[i] != ' ';i++);
  for (i;action[i] == ' ';i++);
  for (i,j=0;action[i] != ' ';i++,j++)
   nom[j]=action[i];
  nom[j]='\0';
  for (i;action[i] == ' ';i++);

  npg=1;
  npd=0;
  for (i,j=0;action[i] != '\0';i++,j++)
  {
   if (action[i] == '(')
     npg++;
   else
      if (action[i] == ')')
        npd++;
   forme[j]=action[i];
   if (npg == npd )
     break;
  }
  forme[j]='\0';

  EvalueForme(forme,eval);

  reg=calloc(1,sizeof(struct registre));
  strcpy(reg->nomreg,nom);
  strcpy(reg->valreg,eval);
  if (regcour == NULL)
   reg->regnext=NULL;
  else
   reg->regnext=regcour;
  regcour=reg;
}
```

```
Addr(action)

char *action;

{ int i,j,npg,npd;
  char nom[20],forme[100],val[200],eval[500];
  struct registre *reg;

  for (i=0;action[i] != ' ';i++);
  for (i;action[i] == ' ';i++);
  for (i,j=0;action[i] != ' ';i++,j++)
   nom[j]=action[i];
  nom[j]='\0';
  for (i;action[i] == ' ';i++);

  npg=1;
  npd=0;
  for (i,j=0;action[i] != '\0';i++,j++)
  {
   if (action[i] == '(')
     npg++;
   else
      if (action[i] == ')')
        npd++;
   forme[j]=action[i];
   if (npg == npd )
     break;
  }
  forme[j]='\0';
  EvalueForme(forme,eval);
  LireReg(nom,val);
  strcat(val," ");     /* ajoute un blanc pour separer val et eval */
  strcat(val,eval);
  reg=calloc(1,sizeof(struct registre));
  strcpy(reg->nomreg,nom);
  strcpy(reg->valreg,val);
  reg->regnext=regcour;
  regcour=reg;
}
```

```
Addl(action)

char *action;

{ int i,j,npg,npd;
  char nom[20],forme[100],val[200],eval[500];
  struct registre *reg;

  for (i=0;action[i] != ' ';i++);
  for (i;action[i] == ' ';i++);
  for (i,j=0;action[i] != ' ';i++,j++)
   nom[j]=action[i];
  nom[j]='\0';
  for (i;action[i] == ' ';i++);

  npg=1;
  npd=0;
  for (i,j=0;action[i] != '\0';i++,j++)
  {
   if (action[i] == '(')
     npg++;
   else
      if (action[i] == ')')
        npd++;
   forme[j]=action[i];
   if (npg == npd )
     break;
  }
  forme[j]='\0';
  EvalueForme(forme,eval);
  LireReg(nom,val);
  strcat(" ",val);     /* ajoute un blanc pour separer val et eval */
  strcat(eval,val);
  reg=calloc(1,sizeof(struct registre));
  strcpy(reg->nomreg,nom);
  strcpy(reg->valreg,val);
  reg->regnext=regcour;
  regcour=reg;
}
```

```
Cond(action,arc)

struct transition *arc;
char *action;

{  int i,j,npg,npd;
   char test[160],subaction[160];
   struct registre *reg;

   for (i=0;action[i] != ' ';i++);
   for (i;action[i] == ' ';i++);
   i++;    /* pour sauter la premiere ( */
   npg=0;
   npd=0;
   for (i,j=0;action[i] != '\0';i++,j++)
   {
    test[j]=action[i];
    if (test[j] == '(')
      npg++;
    else
      if (test[j] == ')')
        npd++;

    if (npg == npd && npg != 0)
      break;
   }
   test[++j]='\0';

   while (Conditions(test) != 0)
   {
    for (i;action[i] != '\n';i++);
    for (++i;action[i] == ' ';i++);
    i++;
    npg=npd=0;
    for (i,j=0;action[i] != '\0';i++,j++)
    {
/* permet de traiter le cas du (T ssi. on a pas d'espace entre la ( et le T */

      test[j]=action[i];
      if (test[j] == '(')
```

```
      npg++;
     else
      if (test[j] == ')')
       npd++;

     if (npg == npd )
      break;
    }
    test[++j]='\0';

  }

  for (i;action[i] != '(' && action[i] != '\0';i++);
  for (i,j=0;action[i] != '\n' && action[i] != '\0';i++,j++)
    subaction[j]=action[i];
  subaction[j]='\0';

  ExecuteUneAction(subaction,arc);

}



Verify(action)

char *action;

{ int i,j;
  char forme[160];
  struct registre *regtamp;

  regtamp=regcour;
  regcour=regdern;
  for (i=0;action[i] != ' ';i++);
  for (i;action[i] == ' ';i++);
  for (i,j=0;action[i] != '\0';i++,j++)
   forme[j]=action[i];
  forme[j]='\0';
  if (Conditions(forme) == 0)
  {
   printf("\nFORME DANS VERIFY ET RETURN(0) : %s\n",forme);
```

```
    regcour=regtamp;
    return(0);
   }
   else
   {
    printf("\nFORME DANS VERIFY ET RETURN(1) : %s\n",forme);
    regcour=regtamp;
    return(1);
   }

}



To(action,arc)

char *action;
struct transition *arc;


{ int i,j;

  for (i=0;action[i] != ' ';i++);
  for (i;action[i] == ' ';i++);
  for (i,j=0;action[i] != '\0' && action[i] != ')';i++,j++)
    arc->etatsuivant[j]=action[i];
  arc->etatsuivant[j]='\0';
}
```